CONTROLLED INFORMATION SHARING IN A COMPUTER UTILITY

Dean Hanawalt Vanderbilt

24 October 1969

PROJECT MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Cambridge Massachusetts 02139

ACKNOWLEDGEMENTS

I would like to thank Project MAC and the National Science Foundation for their financial assistance during my study at M. I. T.

Thanks also to Professor Jack Dennis for his support, encouragement, and helpful criticism, and to Professors R. M. Fano and R. M. Graham for their suggestions and editorial assistance. I wish to thank D. Austin Henderson, Suhas Patil, and Don Slutz for helpful discussions during the formative stages of this research.

Finally, I am very deeply indebted to my wife, Missy, not only for her typing and financial support, but for having constant confidence and faith in me.

CONTROLLED INFORMATION SHARING IN A COMPUTER UTILITY*

Abstract

A computer utility is envisioned as a large, multi-access computer system providing its users with the ability to store information and share its use with other system users. This thesis considers the nature of information sharing and how a computer utility can provide facilities allowing such sharing to take place in a controlled manner.

From a discussion of the goals of a computer utility, a set of requirements for the facilities of the utility is described. A model is developed which presents a method for structuring information. It is shown that the mechanisms of the model preserve certain structural characteristics of the information, and that these properties can be directly related to the requirements regarding the control of shared information. Extensions of the basic model are described which allow more selective types of control, and which remove some of the limitations of the basic model.

*This report reproduces a thesis of the same title submitted to the Electrical Engineering Department, Massachusetts Institute of Technology, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Massachusetts Institute of Technology
Project MAC
545 Technology Square
Cambridge, Massachusetts
02139

Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01), and in part by the National Science Foundation under Contract GJ-432. Reproduction of this report, in whole or in part, is permitted for any purpose of the United States Government.

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Massachusetts Institute of Technology
Project MAC

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP
None

3. REPORT TITLE

Controlled Information Sharing in a Computer Utility

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Ph. D. Thesis, Department of Electrical Engineering

5. AUTHOR(S) *(Last name, first name, initial)*

Dean Hanawalt Vanderbilt

6. REPORT DATE
October 24, 1969

7a. TOTAL NO. OF PAGES
172

7b. NO. OF REFS
14

8a. CONTRACT OR GRANT NO.
Office of Naval Research, Nonr-4102(01)

b. PROJECT NO.
NR-048-189

c. RR 003-09-01

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
MAC TR-67 (THESIS)

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. AVAILABILITY/LIMITATION NOTICES

This document has been approved for public release and sale;
its distribution is unlimited.

11. SUPPLEMENTARY NOTES

None

12. SPONSORING MILITARY ACTIVITY
Advanced Research Projects Agency
3D-200 Pentagon
Washington, D.C. 20301

13. ABSTRACT

A computer utility is envisioned as a large, multi-access computer system providing its users with the ability to store information and share its use with other system users. This thesis considers the nature of information sharing and how a computer utility can provide facilities allowing such sharing to take place in a controlled ner.

From a discussion of the goals of a computer utility, a set of requirements for the facilities of the utility is described. A model is developed which presents a method for structuring information. It is shown that the mechanisms of the model preserve certain structural characteristics of the information, and that these properties can be directly related to the requirements regarding the control of shared information. Extensions of the basic model are described which allow more selective types of control and which remove some of the limitations of the basic model.

14. KEY WORDS

Computers
Machine-aided computers
Multiple-access computers
On-line computers
Real-time computers
Time-sharing
Time-shared computers

DD FORM 1 NOV 65 1473 (M.I.T.)

TABLE OF CONTENTS

L I S T O F F I G U R E S

LIST OF TABLES

CHAPTER 1

A BEGINNING

1.0. Introduction

The development of multi-access computer systems through the principle of time-sharing has greatly broadened the scope of activity to which computers can be applied. The user of such a system can obtain on-line access to the facilities of a large computing system at a small fraction of the cost of a comparable dedicated system. Thus it is practical for him to make frequent, interactive use of a computer system which makes available to him a large variety of computing services. One of the important services, which many systems provide, is the provision of facilities which enable a user to store information within the system and utilize it at some future time.

In their simplest form, these facilities allow a user to store files containing programs or data, and to associate a name with each file by which he may refer to the file. Protection is provided so that each user may access only those files which he requested to be stored. However, in some cases, these facilities have been expanded so that information stored by individual users can be made available to other users.

These developments have led some to predict the evolution of a computer utility. The computer utility is envisioned as a multi-access system with facilities for general use by large numbers of people. In addition to making "computing power" available to users, it would provide a vehicle through which information could be easily shared among users. This would involve both sharing of access to data bases of interest to a number of users and sharing of the use of programs written by users.

It is the objective of this work to provide a greater understanding of the consequences of information sharing. A computer utility must enable users to share, but it must allow them to control the sharing. In the following, the nature of controlled information sharing is considered, and requirements it places on the facilities offered to users of a computer utility are described. An abstract description, i.e. a model, of a computer utility design is described which satisfies the requirements imposed on a utility by the provision of controlled information sharing.

1.1. General Background

A large amount of previous work by others is relevant to the topic being considered. It can be grouped into roughly three classes.

Descriptions of the Computer Utility

A number of authors have discussed the information sharing facilities a computer utility should provide. Fano ([1] - [2]) indicates that an important contribution of the computer utility would be to encourage "system users (to) build upon each other's work." By this he does not just mean that users should store papers in the system, but that the results of work should be directly usable by others. In particular, if the results are a computer program or a data base, the utility should allow them to be stored so that they can be used directly by others. This means that others should be able to use them, by only "interfacing" to them, without understanding their internal operation.

Parkhill [3] also sees shared information created by users to be very significant. He indicates that "public files would constitute its (the utility's) greatest asset," and thus maximum protection from unauthorized access must be provided.

Dennis [4] views the role of the computer utility as providing "an environment in which small information systems may flourish and compete as private enterprises." These information systems would provide their services through the use of programs and data maintained within the utility. Thus, the utility must provide facilities which allow others to take advantage of these services, while still guaranteeing that the proprietary nature of the shared information is protected.

Chapter 2 contains a discussion of requirements which these goals place on facilities offered by the utility.

Existing Access Control Mechanisms

Mechanisms have been developed in various existing multi-access systems to allow information created by users to be made available to other users. The basic viewpoint taken in all of these systems is that a user may store information, in the form of a file, within the system, and that information may normally be accessed only by him. Means are then provided for the users to make the information available to other users.

Two of the simpler examples are the user group and the public file. The user group idea has been used by commercial time-sharing systems, such as the RUSH system [5]. The system users are partitioned into groups, and sharing between groups is not possible. Within a group, however, a user, who knows the name of another group member's file, or perhaps a password associated with the file, may utilize that file by simply specifying the appropriate identification information. Other systems, such as the SHARER system [6] and the Lincoln Laboratory APEX system [7], allow a user to specify that a file he created is "public"; that is, the file may be used (perhaps only in a non-altering way) by any user of the system.

Some of the most extensive existing access control mechanisms are provided by the Project MAC systems, C.T.S.S. [1] and MULTICS [8, 9], and the Cambridge system [10]. Each of these allows the owner of a file to associate with the file a list of access control information. This list may contain names of users and the way in which they may access the file, or passwords which must be given to access the file, or some more complex information. In effect, however, each list contains information which enables the system to decide what access rights any particular user may obtain for the file.

Chapter 2 contains a discussion of the capabilities provided by these facilities, and the extent to which they satisfy the requirements imposed by controlled information sharing.

Use of Structured Information

Also relevant here is a body of theoretical work considering various types of structured information and ways of utilizing that structure.

The closest related work is that of Dennis. The idea of a capability, introduced by Dennis and Van Horn [11] and renamed a pointer in [12], is utilized in this thesis. Many of the concepts regarding the information structures of Dennis [12] are carried over into the particular structures discussed here. Also the concept of "programming generality" described by Dennis [12] is closely related to controlled sharing and building on the work of others. If programming generality is restricted to information stored within a single system, it becomes essentially equivalent to ability to build on the work of others. In fact, the requirements described by Dennis in [12] for a naming scheme (that is a set of rules relating instances of identifiers to stored items of information) to exhibit programming generality are satisfied by the mechanisms for using information structures described in Chapter 3.

The work of Evans and LeClerc [13] also contains indications of a number of ideas used here. Their idea of a "parameter space," indicating an implicit structuring of information, and the idea of relating access abilities to the program being executed are closely related to activation maps and procedures used in the model. However they were concerned primarily with processor design, and thus did not extend their ideas to the organization and accessing of stored information.

Similar models for structured information have appeared elsewhere, such as in the work of Lucas, et al [14] in specifying the semantics of PL/I, but the structure is used for different objectives.

1.2. Overview

The main results of this work are contained in Chapters 2 through 5. In Chapter 2, the consequences of controlled information sharing are considered, and requirements, which the sharing mechanisms of a computer utility must satisfy, are developed. Arguments are also given regarding the inability of existing systems to satisfy these requirements. Chapters 3 and 4 contain the development of a model of structured information, and mechanisms for utilizing that information which allow the requirements in Chapter 2 to be satisfied. Extensions of the model are described in Chapter 5 which allow the user to attain finer degrees of control over shared information than is possible in the original model.

Chapter 6 contains extensions of the model which remove restrictions, on user capabilities to utilize information, which are not directly related to information sharing. Some problems which would be associated with implementing the mechanisms of the model are discussed in Chapter 7. Chapter 8 contains a summary of the ideas used in preceding chapters, conclusions regarding accomplishments of the work, and suggestions for further work based on these accomplishments.

CHAPTER 2

REQUIREMENTS OF INFORMATION SHARING

2.0. Introduction

This chapter contains a discussion of the requirements placed on the facilities of a computer utility by information sharing. The types of activities which users should be able to perform are described, and requirements these activities place on mechanisms provided by the utility are developed. Arguments that mechanisms supplied by existing systems do not satisfy these requirements are given also.

At the end of the chapter, the general requirements discussed in the preceding sections are specified in a more precise manner, using relationships, between users and information, previously defined in the chapter. The arguments concerning the validity of the models developed in Chapters 3 and 4 are based on a demonstration that the requirements developed in this chapter are satisfied.

2.1. Principals and Users

Abilities and responsibilities regarding access to facilities and stored information must be associated with the users of the utility. This is accomplished by postulating a set of abstract entities, called *principals*. These are assumed to exist within the utility, and abilities and responsibilities are associated with them. As a user initiates his use of the system, he becomes identified with a particular principal, and thus obtains the exact abilities associated with that principal.

Use of this abstraction has two advantages. First of all, it is more general than associating abilities with users. A user may perform different roles with respect to the utility, e.g. he may work on different projects, and different people may perform the same role at various times. Thus, a principal can be made to exist for each of these "roles," and the user's abilities will depend on the role he is currently playing. Secondly, use of principals allows separation of the problems of user identification (which will not be of concern to us) from the problems of controlling access to information.

Thus we shall consider controlled sharing of information among principals. It will be assumed that no more than one user may be associated with each principal at any time. This forces the actions of a principal to be performed sequentially, and the effects of concurrent actions by different users need be considered only on an interprincipal basis.

2.2. Use of Shared Information

The activities of a principal (that is, a user associated with a principal) in his utilization of the utility can be separated into two categories. The first consists of those activities involving the use of information to which the principal has access. This might involve executing owned or borrowed programs on data supplied by the principal, or accessing and maintaining a data base, or creating new programs or data. The activities in this category can be characterized by the fact thay they are performed independent of the actions of other principals, except for any interactions resulting from alterable information accessible to more than one principal.

The second category contains those actions which affect the abilities of principals to access stored information. These are actions such as granting others the ability to utilize some particular information, or establishing an access path to information which another principal has shared with him, i.e. granted him the ability to access.

The remainder of this section provides a discussion of some of the access control requirements which are consequences of the first type of activity. The second category is discussed in Section 2.3.

Use of Shared Programs

One type of information which a principal may wish to utilize is a program, owned by another principal, which he has borrowed. Actually it is the _use_ of the program which is being shared, where by _use_ we mean the program's execution. This execution could be requested, in general, directly by the borrower (through the use of a command) or by a program created by the borrower (through a call).

To achieve controlled sharing, the user of a shared program must be able to execute the program without knowledge of its internal structure or operation. Furthermore, the user must be unable to obtain that knowledge without specific provision by the owner of the program. The arguments for this requirement are based on the goals of a computer utility.

As we indicated in Chapter 1, the sharing of information through a computer utility should enable its users to build on the work of others. This means that it must be possible to make use of the results of another's work without necessarily understanding his methods or the steps used in obtaining the results. If the results are in the form of a program, use of the program should be possible without an understanding of its internal operation.

In addition, the computer utility should be a vehicle through which programs, whose structure is of a proprietary nature, may be made available (presumably at a fee) for use by the user community. In this case, users must be able to use a program without it being _possible_ for them to view its internal structure. This need for the ability to execute a program without knowledge of its structure imposes a number of requirements on the organization of shared information and the mechanisms through which it is used.

The execution of a program, in general, involves the use of various data and other programs (subprograms). This additional information can be divided into two categories: that information which must be available for each activation, and that which is associated with a particular activation. The former category (Category I) consists of subprograms which may be called by the program, subprograms of the subprograms, ..., as well as any data which retain values from one activation to the next. Category II consists of the information passed, e.g. as call arguments, to and from the initiator of the program, as well as temporary information used by the program during its execution.

In order to allow the execution of a program, use of this additional information must be provided during program execution. There are two aspects to this use. First, the names used by the program to refer to this additional information must be bound to the correct information. Secondly, the access control mechanisms of the utility must allow access to the information when it is needed.

The Category I information is known to the owner, i.e. the creator, of the program, but not to the borrower. Thus the owner must specify the binding of names in the program to that information, and ensure that the information is available when needed during an execution of the program. Since the program borrower should be granted no more access abilities than necessary, it must be possible for the owner to give the borrower the ability to access the information <u>only</u> in conjunction with use of the program. Thus, access abilities and binding information must be associated with the shared <u>program</u> so that the appropriate Category I information is available each time the program is executed.

The Category II information consists of information supplied by the program user and information created by the program. For the former, the supplied arguments must be bound to the program's call parameter names. The access abilities pose no problem since this is information belonging to the program user. For the latter information, the process executing the program must be allowed to "create" information and to have it automatically bound to the appropriate names appearing in the program.

In the next section, the ability of existing facilities to provide for the program sharing is discussed relative to these requirements.

Existing Sharing Mechanisms

Existing systems, in which information sharing is possible, allow users to store information in the form of files. A file may consist of a program (or a set of related programs) or data. Generally each user has a directory which associates a symbolic name with each file owned by him, and he refers to the file by this name. Each of the directory entries must have a different name so that its name identifies a file uniquely. Depending on the particular system, the owner may allow other users to access a file in a number of ways. He may specify to the system the names of the allowed borrowers, or may specify a password which must be known by a borrower, or he may just allow anyone who knows its name to access it. In any case, the borrower establishes his ability to access the file by "linking" to it. This results in an entry being formed in the borrower's directory which associates a name of his choosing with the borrowed file. The borrower is then able to utilize the file in the same manner as his own, possibly subject to some restriction such as inability to alter the file.

When a user requests the execution of a program file to which he has linked, the Category I information required by the program is found by matching names appearing in the program to entry names in the user's directory or in a directory of publicly available files, e.g. library routines. The Category II information supplied by the user to the program is also specified by entry names in his directory, and files created during execution of the program are entered into the user's directory. Unfortunately, these mechanisms are not adequate to accomplish the controlled sharing described in the preceding section.

First of all, entries for the Category I information required by a shared program must exist in the borrower's directory or a public directory. Thus the borrower must know of this information, and must have established his ability to access it. This means that the borrower must have the ability to utilize the Category I information at any time in order to have it available for use with the shared program, and thus has more access ability than is necessary to accomplish the sharing.

Secondly, files created during a user's execution of a program are entered into theusers's directory with entry names chosen by the program. Thus the borrower of a program must be sure that none of the file names used by the program or its subprograms conflict with existing entries in his directory. Also the creator of a program must be certain that file names used by its subprograms, which may be borrowed, do not conflict with each other or with names used by the main program.

In each case, the user of a shared program is required to have knowledge, about the program's internal operation, which should not be necessary for use of the program.

The use of multiple directories organized in a tree-like structure, such as in the MULTICS system, can be helpful in resolving name ambiguities. Different directories can be used for the information used with different programs, and the order in which the directories are searched can be altered. Thus, a user can have the ability to access a number of different files with the same name, and have each used at the appropriate time by changing the directory search strategy. However neither of the above problems are overcome, since access abilities are still associated with users and search strategies are specified by the program user, not the creator.

It should be noted that the MULTICS system does allow, to a limited extent, the association of information with a program. This can be accomplished by creating a process which has the ability to access a program and information associated with it. No other processes are allowed to utilize the program or its associated information except indirectly through communication with the process. This mechanism is used, for example, to protect shared data bases by creating them as information private to a particular process which executes only the programs which maintain the data.

This is not a general solution to the problem, however. The problem of associating programs, along with their associated information, with other programs is transformed into associating, with a process, the ability to communicate with, i.e. access, other processes. Thus, the same naming and access control problems appear in an interprocess context which were just described as problems in associating programs, and which also are not solved in existing systems.

Shared Data

We have considered requirements which shared programs place on facilities of the computer utility. What about shared data? Again it is the use of the data which is to be shared.

We require that access to shared data be allowed only through a (caretaker) program. To share data with others, a principal must make available to others a program with which the data is associated, i.e. a program for which the data is Category I information. A borrower can then establish his ability to use the program, and the program will access the data on his behalf.

Arguments for this requirement are based on the goals for the computer utility described in Chapter 1. Here, "building on the work of others" implies that the user should be isolated from the techniques used to manage the data. Under management, we include such things as the internal structure of the data and the coordination of usage so that only meaningful data alterations and accesses are made. Protection of personal or proprietary information also is important. In the case of a data base, it should be possible for the owner to enforce selective restrictions, so that certain parts of the data entries are available only to designated users, whereas other parts are generally available.

This would be important, for example, if a data base containing personal information is to be used for statistical purposes. Certain sensitive information could be protected while the remainder would be made generally available.

All of these functions can be carried out by an associated, caretaker program, and thus we require that each data base be accessed through a program. This requirement also reduces data sharing to a particular type of program sharing, so that the remainder of the thesis will consider information sharing to be sharing of the use of programs.

It should be noted that existing systems all allow the direct sharing of data. However, this has been done because of efficiency considerations, and not because of any basic need for direct data sharing. We are not concerned here with questions of efficiency, but rather the consequences of user requirements on system organization. Thus, direct data sharing will not be considered.

2.3. Sharing Methods

We have seen in Section 2.2. that controlled information sharing imposes a number of requirements on the facilities through which users make use of accessible information, in particular, shared information. In this section, the effects of this requirement on the other type of user activity are considered. In particular, we consider what types of facilities should be provided for making information available to others and for establishing access to shared information.

As we have argued, only the use of programs should be shared. However, it must also be possible to associate other (Category I) information so that this information is available as the program is executed. Conceptually then, it must be possible to associate access abilities with programs. To differentiate between the actual code of the program and a program along with its associated access abilities, we shall use program to mean the former and procedure to mean the latter. Thus it is use of procedures which is to be shared.

Sharing of procedures implies that a principal may have two types of access ability for information. The first type, called direct ability, applies to those procedures whose execution the principal may explicitly request, and to that data which he may supply to those procedures. Directly accessible information thus includes the principal's own data and procedures, and shared procedures owned by others. The other type of ability, called indirect ability, refers to the additional (Category I) information, both data and procedures, which is associated with the shared, directly accessible procedures, and may be used only through use of those procedures.

Another way in which a principal can utilize information is to create procedures. This would involve 1) the creation of a program, and 2) the association of the program with abilities to access other information. Clearly the only candidates for inclusion within a procedure created by a principal are data and procedures directly accessible to that principal, for those are the only ones of which he is aware. Thus the concept of direct accessibility must be extended so that a principal may create a procedure by associating only directly accessible information with the procedure's program. Note that although indirectly accessible information cannot be explicitly incorporated into the procedure, use of such information is required to execute the procedure. For example, if A is an indirectly accessible procedure whose use is required by a directly accessible procedure B, then any procedure C into which the ability to access B is incorporated must implicitly contain an ability to use A (when it is required by B).

Existing systems except for the use of separate processes in the MULTICS system, allow only direct access to information. If a system allows access abilities to be associated only with principals, either a principal has direct access or no access to information. It is the concept of associating access abilities with programs which leads to indirect access.

Let us now consider the actions involved in sharing information. By sharing we mean the activity which results in one principal obtaining direct access to information created (and owned) by another. There are two aspects to sharing. First, the information which a principal is allowed to make available to others, and, second, the mechanisms used to accomplish the sharing.

Shareable Information

As we argued in Section 2.2., only procedures should be shared, where by a procedure we mean a program and associated access abilities for its Category I information.

Which of a principal's accessible procedures may be shared? We require that a principal may not share borrowed procedures. A borrowed procedure is one which was created, and therefore is owned, by another principal, and which was shared with the borrower. Sharing of a borrowed procedure would not be necessary to make any of the borrower's work available, since he has added nothing to the procedure. Thus the ability to decide with whom and how the procedure is shared should remain with its owner. A principal may then share only owned procedures.

A question remains, however, of which owned procedures may be shared. This question is not trivial since an owned procedure may make use of Category I information not owned by the procedure owner. Thus if a principal P grants direct access to a borrower Q for a procedure owned by P, this may also allow Q indirect access to information owned by neither P nor Q. There are a number of possible ways to resolve this issue.

One approach would deny a principal the ability to share any procedure requiring the use of a borrowed procedure. This would be essentially equivalent to the approach of existing systems we have described. In fact, if all of the information utilized by a procedure is owned by a single principal, he could relatively easily avoid any naming ambiguities, and could collect all of the information into a single file. In that case, then, existing mechanisms would be adequate as they are.

This approach, however, is not consistent with the desire to "build on the work of others" in a controlled manner. For example, consider the situation in which a principal P_2 has been granted direct access to a procedure A owned by P_1, and P_2 desires to share an owned procedure B, which uses A, with P_3. Under the approach just described, this sharing could be allowed only by 1) enabling P_2 to obtain a copy of A, which would then be owned by him and could be shared, or 2) having P_1 grant P_3 direct access to A, and then enabling P_3 to use B and A together somehow. In either case, P_1 is being forced to give another principal more ability than is necessary to accomplish the desired sharing.

A second approach would allow a principal to share all owned procedures. To allow this, the granting of direct access to a procedure would have to imply the ability to grant other principals indirect ability for the procedure. This would enable the borrower to associate the ability to use the procedure with any program which he may create independent of who may come to use the program. Sharing would proceed with each principal able to control who may directly use his work, but not restricting the "propagation" of its (indirect) use to other principals.

Other approaches, lying between these two extremes, are possible. They would allow principals some type of selectivity in the restrictions placed on the "propagation" of use of procedures they have shared. A general discussion of these approaches is given in Chapter 5.

We shall see that the model developed in this thesis will accommodate any of these approaches. However, in order to develop the model in the simplest way, mechanisms first are described which implement the second, unrestricted approach. Mechanisms implementing other possible approaches are then developed as extensions to this original model. The reason for this is that the capabilities needed to satisfy the requirements discussed in Section 2.2. naturally lead to access control of the "unrestricted" type, and the extensions are in the direction of introducing more restrictive control of user activity.

Sharing Mechanisms

One final topic deserves brief mention here, and that is the basic nature of the sharing mechanisms. There are two types of activity which make up the sharing process. First the owner of a procedure determines that it shall be shared, and makes it "available" to others. Second, a borrower establishes his right to access the procedure, and obtains the appropriate access ability. In the preceding section, we were concerned with which procedures the owner may share. We have not considered, however, how a borrower's access rights are determined.

Two approaches to this have been used in existing systems. The first associates with the shared information a list of users and the access privileges they may acquire. The system then determines a borrower's rights by examining this list and acting accordingly. The second approach is to associate a key (perhaps just its name) with the information. A borrower's abilities are then determined by his knowledge of the key.

The first approach, since it implies decisions based on principal identity, results in assured control; that is, it does not rely on information residing external to the utility to make access decisions (assuming that a user has been identified and associated with a principal). This approach will be used in the model development of the thesis.

However, both of these approaches represent special cases of a more general decision method. The general method allows an owner to specify an algorithm by which a potential borrower's abilities could be determined. The algorithms for the above approaches would amount to the matching of a principal's identity to a list of names, or the examination of a proposed "key" supplied by the borrower. Because of its generality, an extension of the basic model implementing this approach is given in Chapter 4.

2.4. Additional Assumptions

Some additional assumptions have been made in development of the model which should be discussed in this chapter. One assumption concerns the alterability of programs, and a second specifies the nature of the sharing agreements made between principals.

The first assumption is that once information is changed into an unalterable form it is never again altered. In the model, this change occurs to a program segment when it is associated with other information to form a procedure and to data which is to be used, in an unalterable way, in conjunction with a program. (Note that data which may be altered by some users and accessed in only a read-only manner by others remains in an alterable form, and the caretaker program provides the appropriate protection.) Since any read-only data used by a program can be considered to be "constants" which are part of the program, this assumption reduces to the unchangability of programs used in procedures.

Programs can be changed into an unalterable form, since we require that programs exist in pure form, and therefore are not altered during execution. This allows re-entrant use and sharing of the use of a program to be accomplished using a single copy of the program. In addition, it ensures a separation of the program from information created during its execution which should be associated just with the particular program activation.

If the use of a program (as part of a procedure) has been shared with others, the owner should not be able to alter the program. The borrowers will in general have used the program as a subprogram for procedures, use of which may have propagated to additional users and use in other procedures. This may lead to certain problems if program alteration is allowed. First, since it is generally difficult or impossible to establish the equivalence of two programs, the introduction of an altered, "almost equivalent" version of a program may affect the operation of the procedures in which it is used. Secondly, it may be quite difficult to determine if and by whom a program is being used or may be used at any given time. Since a program cannot be altered while it is being executed, choosing a time to alter it is quite difficult.

Thus we are reduced to the case in which a program is part of a procedure accessible only to its owner. This would occur if use of the procedure has never been shared or if all borrowers had relinquished their capability to use the program. (This latter condition might also be difficult to determine.) In this case, alterations to the program must be performed by the owner. Thus it seems reasonable that the owner be forced to recognize that the altered program is different by constructing it (perhaps by copying much of the original program) as a logical entity different from the original program.

In no cases, then, will programs be altered. In order to "replace" a program within a procedure, each of the subprocedures of the procedure which (directly or indirectly) make use of the program must be reconstructed by their creators. This ensures that the changed procedure is recognized as being different from the original, and that each contributor to the construction of the procedure, who might be affected by the changed program, is forced to recognize that a change has occurred. (In Chapter 5, we shall see that this process need not be too tedious if replacement is all that is required!)

The second assumption is that, under normal circumstances, a user loses abilities only through his own action. Thus, a user is the only one able to access his owned information unless he chooses to make it available to others. In addition, once a borrower has gained access to shared information, that ability will not be lost unless it is relinquished by the borrower. The reason for this is that the borrower will in general have done work, such as constructing other procedures and offering them to others or embarking on a reserach program, which depends on the availability of the shared information, and the owner should therefore not be able to arbitrarily remove an access privilege already granted. We require, then, that the establishment of the borrower's access ability represents the formation of a contract which binds the owner to provide use of the information.

Of course there may be reasons for termination of the contract, i.e. removal of the access ability. The borrower may fail to provide the compensation to the owner specified in the contract, or it may be discovered that the shared information fails to meet the promised specifications. In either case, however, this would require a decision by a higher authority. Thus there must be capabilities within the utility for determining, in response to user request, if a contract violation has occurred, and for changing the appropriate access abilities. Since these capabilities might in general depend on legal and other considerations outside of the scope of this research, they will be considered only with regard to the means required to change established access abilities.

2.5. Model Requirements

The following two chapters contain the development of a mathematical model. The objective of this development is to describe an approach for organizing and utilizing stored information, and to demonstrate that this approach allows controlled information sharing to be accomplished. In this section, we summarize the requirements which the model must satisfy, based on arguments given in the preceding sections of this chapter.

We have argued that only the use of procedures should be shared. Also principals must be able to make use of information accessible to them without affecting the information accessible to others, except for indirect effects due to alteration of data associated with shared procedures.

The types of activity which a principal can perform using accessible information are to request the execution of procedures and the creation of new procedures. As part of a request for a procedure execution, the (Category II) information to be supplied as arguments to the procedure's program must be specified. This supplied information can be any information which is accessible to the principal. All information remaining following a procedure execution, which is not part of the procedure, becomes accessible to the principal requesting the procedure execution. Thus, creation or alteration of accessible data can be accomplished by a procedure execution.

A principal can create a procedure by associating an accessible program with other accessible information. Creation of new procedures can involve the association of alterable data with a program. In order to ensure orderly use of the data, the principal creating the procedure must be restricted henceforth to access the data through the procedure, as must the others with whom the procedure is shared.

Using the concepts of direct and indirect accessibility, we can summarize these requirements as:

Requirement 1. Only procedures can be directly accessible to more than one principal.

Requirement 2. A principal can request the execution of any directly accessible procedure, and can supply, as arguments to the program of the procedure, any directly accessible information of the appropriate form. It is possible for information to be created during the execution of a procedure, and any of that information, which is not destroyed during execution, becomes directly accessible to the principal requesting the execution.

Requirement 3. A principal can create a procedure using any information directly accessible to him. Any alterable data associated with the procedure must become indirectly accessible (through the procedure) to the principal.

Requirement 4. Execution or creation of a procedure by a principal cannot affect the directly accessible information of any other principal, except through the alteration of data indirectly accessible to both principals.

The execution of a procedure is accomplished through an activation of the program part of the procedure. This involves associating values with the names appearing in the program which are assumed to be predefined, and then executing the program.

It is useful to think of the execution of a program being carried our by a process, or a "locus of control", which accomplishes the actions specified by the program instructions. In the model development, we assume that just one process is used in the execution of a procedure. (A generalization of the model to encompass parallel processing is discussed in Chapter 6.) While the process is executing a program, it must be able to use just the information associated with the program by the procedure and the information supplied or created for this particular activation, and it must be able to use this information only in the intended manner. For example, in the case of a subprocedure (an associated procedure), the process must be able to cause the subprocedure's program to be executed, but it must do this while ensuring that information regarding the internal structure of the subprocedure cannot be acquired during execution of the main program.

To help make this discussion more precise, we differentiate two types of information associated with a program by a procedure. The data and procedures intended to be explicitly used by the program, and thus for which the program contains names, are said to be directly associated with the program by the procedure. The data and procedures associated with programs of directly associated procedures are <u>indirectly associated</u> with the program, as is information associated with the programs of these indirectly associated procedures, ... Thus the data and procedures whose use is required by the program and which is specified by the procedure creator are directly accessible to the program, and the information required by any of those procedures is indirectly associated.

We can now specify requirements for the operation of a process in executing a procedure:

<u>Requirement 5.</u> During execution of the program of a procedure, P, a process can use all data directly associated with the program of P, supplied by the initiator of the program execution (either a principal or another procedure), or created during the program's execution. Furthermore, the process can create and use additional data.

<u>Requirement 6.</u> During execution of the program of a procedure P, a process can request the execution of any procedure directly associated with the program by P. However, while executing the program of P, the process cannot access the program of a directly associated procedure or gain any knowledge of its directly associated information. The same requirements are true for procedures supplied as arguments for the particular activation of the program of P.

Requirement 5 specifies that all of the data intended for use with the program of a procedure be made available to the process although only part of it may have been supplied by the principal or program requesting the application of P. Requirement 6 constrains the use of one procedure in another so that the requirements regarding the protection of proprietary information are satisfied.

The other requirements for the model concern the activity of sharing procedures between principals. For the development in the next two chapters, a relatively unrestricted type of sharing is utilized. (Chapter 5 describes other types of sharing restrictions and how they might be implemented.) Using the concept of ownership which we have discussed in this chapter, the requirements on the model's sharing mechanisms are the following:

Requirement 7. Each procedure created by a principal becomes owned by him.

Requirement 8. A principal can make the use of any owned procedure available to any principal or group of principals.

Requirement 9. A principal can establish direct access to any procedure made available to him.

CHAPTER 3

INFORMATION STRUCTURES AND PROCESSES

3.0. Introduction

We have discussed the relationships of ownership and direct and indirect accessibility, which exist between principals and information, and of direct association, existing between programs and other information. In the requirements described in Section 2.5., these relationships are extremely important in determining the ways in which information can be utilized. In fact, they are given meaning by the requirements.

The approach of this model is to specify a structured form for stored information, in which the structure reflects these relationships, and to specify mechanisms for utilizing structured information. Because of this correspondence between the structure of information and the ways it can be utilized, the requirements of Section 2.5. can be restated in terms of structural properties of the model. Thus, by demonstrating that the mechanisms of the model preserve certain structural properties, they can be shown to satisfy the requirements of controlled information sharing.

In this chapter, we introduce the idea of structured information and describe the portions of the model concerned with procedure execution. First, structures called elementary structures are defined. These can be used to represent procedures having no associated alterable information and utilizing Category II information which is composed solely of alterable data. A description is given of the means, for identifying and accessing information, used by a process while executing one of these procedures. Some properties of these mechanisms are noted and related to the model requirements regarding procedure execution. The final section of the chapter contains a generalization, of the elementary structures, allowing alterable data to be associated with procedures, and procedures and read-only data to be specified as Category II information for a procedure activation.

The structures and mechanisms developed in this chapter are used in the complete model, which is described in Chapter 4. The complete model embeds these ideas in a representation of all of the information stored within a utility and mechanisms for using it.

3.1. Information Structures

Information is stored and utilized in the form of structures. Information structures are acyclic, connected, directed graphs composed of two classes of nodes, called segments and connectors, with directed arcs, called branches, between them. Segments must be terminal* nodes, non-terminal* nodes must be connectors, and each structure must have a unique root* node which is a connector. Associated with each branch of a structure is a symbolic name. Branch names must obey the rule that no two branches leaving the same node have the same name. Each node has an associated type, which must be E, R, or W. Figure 3.1. shows an example of an information structure, in which connectors are represented by rectangles and segments by ovals. Each node contains a unique number by which it can be identified, e.g. node 1 is the root node.

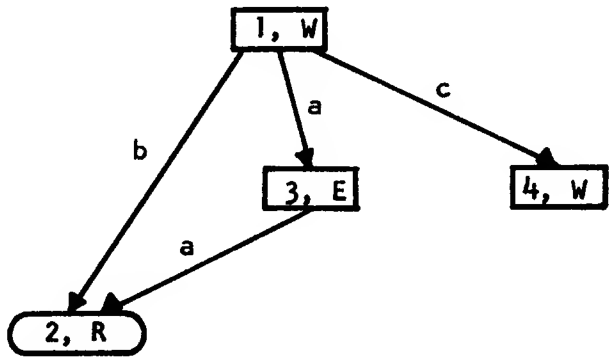


Figure 3.1. An Information Structure

* A terminal node has no branches leaving it, a non-terminal node has branches leaving it, and a root node has no branches terminating on it.

Each segment of an information structure is an array of words; the array can be of arbitrary length. A segment may contain, for example, a program or a set of related programs, or it may contain a set of data. We shall assume that the contents of a segment are equivalent with respect to access control. That is, the finest level at which access abilities may be differentiated is the segment level. If two principals, or two processes (as we shall see later), have the same access ability for any part of a segment, each has the same ability for the entire segment.

The branches of information structures represent abilities to access information. If an entity, e.g. a principal, is able to access the node from which a branch emanates, then the entity may also access the node on which the branch terminates. Since there may be many branches leaving a node, however, a means for differentiating among them must be provided. Use of the branch name accomplishes this, since it is required that all branches leaving a node must have different names. The way in which a node may be used is specified by the node type. This constrains the type of access ability to be Execute, Read-only, or Write and Read. (The precise meaning of these types will be discussed below.)

The role played by connectors is to allow associations of access abilities to be formed. From the definition of an information structure, segments must be terminal nodes. Thus branches may emanate only from connectors. As we have described, ability to access a non-terminal node, i.e. a connector, implies the ability to access the nodes to which the branches lead.

Of course, some of the branches leaving a connector may terminate on connectors which have branches leaving them. Ability to access the original connector implies ability to access the intermediate connectors, and those abilities in turn imply abilities to access the succeeding nodes. Continuing this argument, then, we see that ability to access a connector implies the ability, subject to node type restrictions, to access all nodes on "paths" of branches from the connector. Thus a connector can be thought of as defining a structure, which is composed of all nodes on brancn paths leaving the connector, and access to this root connector of the structure can be thought of as access to the structure.

3.2. Elementary Information Structures

In this section, we describe some particular classes of information structures, whose form is made to reflect relationships among the information contained in the structure. As a vehicle for discussion, we use the following example.

A principal P is constructing a table-driven compiler. The compiler is composed of 1) a program segment, which reads from 2) a set of tables in a data segment, and invokes the use of 3) another program segment S as a subroutine. Principal P would like to associate segments 2 and 3 with program 1 to form a procedure as shown in Figure 3.2.(a). The main program 1 would have associated with it the abilities to access 2 and 3, which are represented by the arrows leaving 1. During the execution of 1 the other segments could be referenced by symbolic names, say Table and Subr, and the named arrows leaving node 1 would indicate the particular segments being referenced. Of course, during the compiler's execution, the two program segments would be executed and the table would be read and not altered. This is indicated by the notation appearing in the nodes of Figure 3.2.(a).

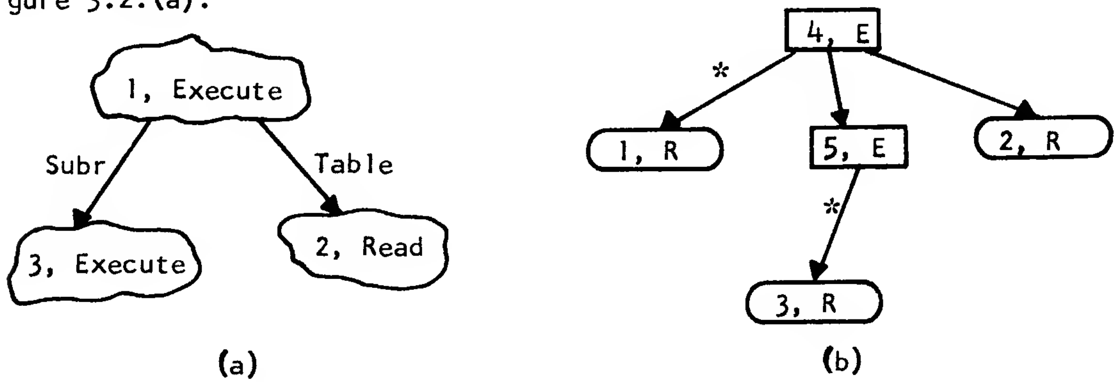


Figure 3.2. Evolution of an Elementary Procedure Structure

Elemenatry Procedure Structures

We must be able to represent this procedure as an information structure. However, segments and branches cannot be directly substituted into the structure in 3.2.(a), since segments must be terminal nodes and only connectors can be used to associate information. This difficulty can be overcome quite simply, by giving special significance to certain branch names.

The branch names of each information structure are taken from a set $A \cup \{*\}$, where A is countable and $* \notin A$. We say a branch with name $*$ is a <u>*-branch</u> and one with a name from A is an <u>A-branch</u>.

The method of representing a procedure by an information structure is relatively straightforward. A *-branch leaves the root node (a connector) and terminates on the program of the procedure. A-branches leave the root node and terminate on information which is directly associated with the program by the procedure. The root node is of type E and the program and read-only data segments are of type R. The precise significance of the node types is discussed in the next section. Intuitively, however, the type E root node indicates that the procedure may only be executed, and the type R segments indicate that, during execution, the segments are read but not altered. (As we indicated in Chapter 2, programs are in pure form and thus are not altered during use.)

Returning to the above example, the information structure shown in Figure 3.2.(b) represents the procedure described by 3.2.(a). Note that the subroutine 3 was transformed into a procedure structure in which no other information is associated with the program.

This type of procedure structure, which associates with a program only abilities for read-only segments and other procedure structures of the same type, is called an elementary procedure structure. This type of structure can be more formally defined by:

Definition 3.1. The class of elementary procedure structures is the set of all information structures satisfying the following property:

> The root node of the structure is a connector of type E having a *-branch leaving it which terminates on a segment of type R, and zero or more A-branches leaving it, each with a different name, such that each terminates on a type R segment or the root node of an elementary procedure structure.

We abbreviate elementary procedure structure by eps.

Elementary Data Structures

In order to make use of the compiler, principal P must have a means for supplying data, namely the source program, to the procedure and for receiving the compiled object program. In addition, since the compiler programs are pure, the procedure must be given the ability to create working space for the storage of temporary data, such as the values of internal variables and tables. Both of these requirements can be met by the use of another type of structure, the data structure.

A data structure contains all of the information, used by the process for an activation of the program of a procedure, which is not contained in the procedure structure. When execution of a procedure is requested, a data structure is specified which contains the information to be used as arguments for the requested activation of the procedure's program. The process may expand the data structure during execution of the program to accommodate both the "output" from the procedure and temporary storage space. When the execution is completed, the data structure, containing "output" from the execution, is returned to the initiator of the procedure execution.

A simple form of data structure is the elementary data structure. It is an information structure in tree form, containing A-branches, type W connectors, and segments. In the case of elementary procedure structures, we did not rule out shared substructures. Figure 3.3. shows an eps with two subprocedures both containing a third subprocedure.

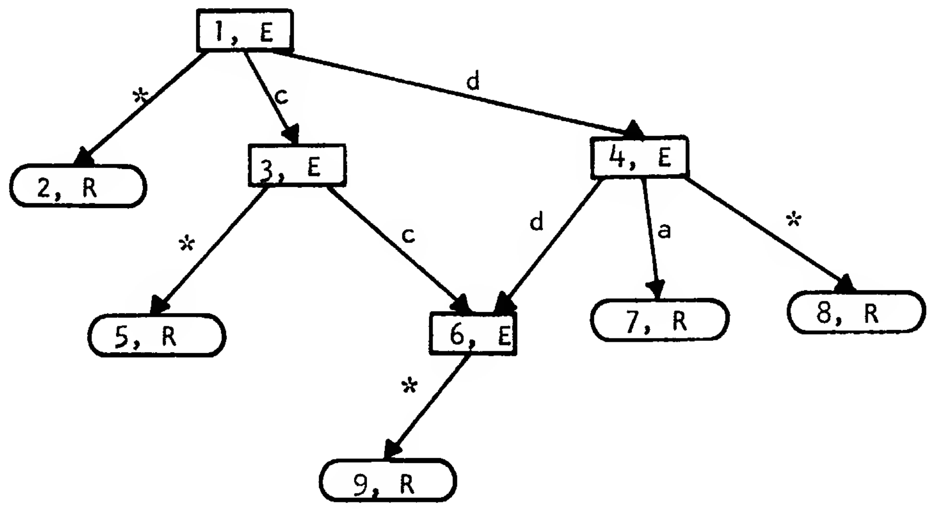


Figure 3.3. An eps with a Shared Subprocedure

This is allowable since the subprocedure is not altered during execution, and the operation of the entire procedure is the same as if separate copies of the shared procedure were contained in each of the sharing procedures. However, in the case of data structures, having more than one "path" to a type W node could lead to inconsistencies. Thus, we require that each type W node of a data structure have no more than one branch in the structure terminating on it.

Definition 3.2. The class of elementary data structures is the set of all information structures satisfying the following property:

> The root connector of the structure is of type W and has zero or more A-branches, each with a different name, leaving it with each branch terminating on a segment of type W or R or a root node of an elementary data structure. In addition, each node of the structure must have no more than one branch of the structure terminating on it.

We shall abbreviate elementary data structure by eds. An example of an elementary data structure is shown in Figure 3.4.

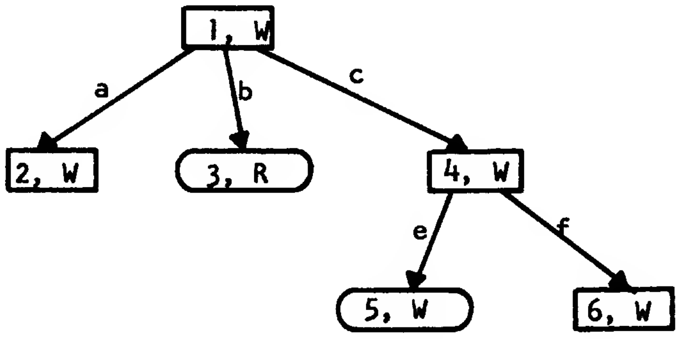


Figure 3.4. An Elementary Data Structure

3.3. Pointers, Activation Maps, and Processes

In the previous section, two classes of information structures were defined. In order to see how these structures can be used to provide the controlled access to and utilization of information, we must discuss the mechanisms used to "execute" the procedure structures.

We assume that all information structures are stored in a Storage Subsystem. The Storage Subsystem associates a unique identifier with each information structure node. Requests for access to nodes in the Storage Subsystem must be accompanied by a token for each node specified. Each of these tokens, which we call pointers, consists of two components: a node identifier and a specification of the access ability it provides. This access ability is specified in two parts, the first being the node class (segment or connector) and the second being an access mode restriction, which may be Execute (E), Read-only (R), or Write and Read (W). The significance of this second pointer component is discussed later in this section.

Definition 3.3. A pointer is an ordered pair (i, a), where

i is a node identifier, and

a is an ordered pair (c, t),

where $c \in \{$segment, connector$\}$ and

$t \in \{E, R, W\}$.

The second component of a pointer is called its type. Its value will be abbreviated by the first letter of the node class (S or C) followed by the access mode restriction, e.g. SE, CW, ...

The activity of executing programs is carried out by a Processing Subsystem. It contains a number of processes, each of which is concerned with the execution of a program, or more exactly, the program of a particular procedure structure. Within a program, there exists a set of symbolic names which denote variables. For an activation of a program, these variables take on particular values, and these values are utilized as specified by the instructions appearing in the program. Since it is necessary for a program to refer to nodes of information structures, some of these variables take on values which are pointers. Since these values determine the access abilities associated with the program activation, the model is concerned only with variables which take on pointer values.

<u>Definition 3.4.</u> Let $\overline{X} = \{x_0, x_1, x_2, \ldots\}$ be a countable set from which all variable names are taken. Then an <u>activation map</u> is a finite set $X \subset \overline{X}$ of variable names, and a partial function f which maps some (possibly all) of the names in X into pointer values.

The set X represents all variables appearing in the program which take on pointer values, and the function f represents the correspondence between variables and pointers. In general f is a partial function, since at any point in the execution of a program, not all of the variables appearing in the program are defined, i.e. have values.

The interaction between the processing and storage subsystems, can be depicted by Figure 3.5.

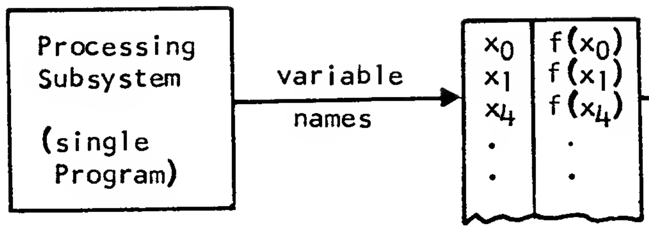


Figure 3.5. Use of Activation Map

The execution of a program in the Processing Subsystem causes references to variables, which are transformed through an activation map into pointer values which are presented to the Storage Subsystem. Of course, the execution of a program may require the execution of other programs, i.e. the programs of directly associated procedures, which may themselves have associated procedures. During the execution of each of these programs, a different activation map must be in effect since each program may associate its own meaning with the variable names it uses. Also to guarantee the required "isolation" between a program and procedures it may use, the utilization of a different activation map for each program restricts access by the subroutine to only that information required for its execution.

As we indicated in Chapter 2, a program is thought of as being executed by a process, a locus of control which performs the actions necessary to "execute" a procedure structure. We assume that the mechanism for requesting, in a program, the execution of a subprocedure is a call-return mechanism (whose detailed specification will be described shortly). If during the execution of a program, a process is required to make use of a subprocedure through a call, a new activation map is created for the program of the subprocedure, and the process begins execution of the program using that map. When that execution is completed, the process returns to continue execution of the calling program using the map which was in effect for that program, and the activation map for the subprocedure ceases to exist.

Thus the access abilities of a process can be described by a sequence of activation maps, the last in the sequence being "currently in effect."

Definition 3.5. A process state is a finite sequence of activation maps, $(M_1, M_2, \ldots, M_k)$.

We can thus see that the interaction between processing and storage can be represented by Figure 3.6.

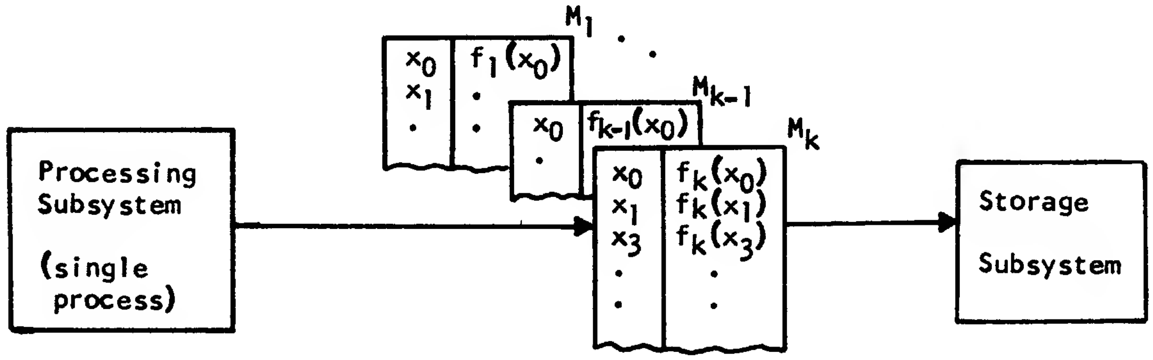


Figure 3.6. Use of a Process State

As we have indicated, the access abilities of a process vary during the course of its activity. The values associated with variables are changed as nodes are accessed; activation maps are created and destroyed as control passes between programs. In the next section, we describe a set of operations, which a process may perform while executing a program, which allow the process to utilize the information associated with the program being executed. These operations are described in terms of their effect on the state of the process, and properties of these operations are discussed relative to the requirements discussed in Chapter 2.

3.4. Program Instructions for Elementary Structures

As a process executes the program of a procedure, it must be able to utilize the information directly associated with it in the procedure and other information associated with the program only for that particular activation. This information is contained in the procedure structure of the program being executed and the data structure available to the process. Thus, it must be possible for the process, using the activation map associated with the program activation, to access components of these structures.

The method used to implement this provides an activation map for a new program activation, which associates, with standard variable names, pointers to the program to be executed, to the root node of its procedure structure, and to the root node of the data structure to be used for this activation. The program contains instructions which enable the process to access other nodes of these structures, by traversing branches which leave nodes already accessed, and to create and delete portions of the data structure.

When a process is initiated, the activation map M_1 is established with values assigned to names x_0, x_1, and x_2. The value $f_1(x_0)$ is a type SR pointer to the program of the procedure structure being activated, $f_1(x_1)$ is a type CR pointer to the root node of the procedure structure, and $f_1(x_2)$ is a type CW pointer to the root node of the data structure to be used by the process.

The instructions which can be used by programs to make use of the structures available to it are described in Table 3.1. It is assumed

TABLE 3.1.

Assume that the process is using activation map M_k when the instruction is executed.

1) x_j = <u>obtain</u> <u>node</u> (x_i, b) - If $f_k(x_i)$ is a pointer of type CR or CW, and a branch named b leaves the connector, $f_k(x_j)$ becomes a pointer to the node on which the branch terminates. The type of $f_k(x_j)$ is specified by the class and type of the newly accessed node.

2) x_j = <u>create</u> <u>node</u> $\left(x_i, b, \left\{\begin{matrix}\text{segment}\\ \text{connector}\end{matrix}\right\}\right)$ - If $f_k(x_i)$ is a type CW pointer and no branch with name b leaves it, a branch with name b is established from the commector to a newly created node, which is of type W and of class specified by the third parameter. $f_k(x_j)$ is established as a pointer to the newly created node, with its type specified by that node's class and type.

3) <u>call</u> (x_i, x_j) - If $f_k(x_i)$ is a type CE pointer and $f_k(x_j)$ is a pointer of type CW, a new activation map M_{k+1} is established such that $f_{k+1}(x_1)$ is a type CR pointer for the node specified by $f_k(x_i)$, $f_{k+1}(x_0)$ is a type SR pointer for the program segment of the called eps (specified by $f_k(x_i)$), and $f_{k+1}(x_2) = f_k(x_j)$.

4) <u>return</u> - If $k>1$, the activation map M_k is deleted and M_{k-1} is reactivated. If $k=1$, this signals completion of the task and the process is terminated.

5) <u>delete</u> (x_i, b) - If $f_k(x_i)$ is a pointer of type CW to a connector with a branch with name b leaving it, that branch is deleted from the structure.

that the process is executing the program containing the instructions using activation map M_k. The form of each instruction is shown, and a description of the instruction's effect on the process state is given. The symbols x_i and x_j are used as variable names and b is used as a branch name such that $b \in A$.

The instructions _obtain node_ and _create node_ allow access to other nodes of the procedure and data structures to be gained by branch traversals from nodes already accessed. In the case of _obtain node_, the newly accessed node already exists, whereas _create node_ causes a new branch and a new node to be created. In both cases, the variable specified on the left hand side of the instruction takes as value a pointer to the newly accessed node, with the pointer type being specified by the accessed node's class and type.

The instructions _call_ and _return_ allow control to be transfered between programs. Use of _call_ causes an activation map for the program of a directly accessible procedure to be created with initial variable values assigned, and execution of _return_ causes the activation map to be deleted and control to be returned to the calling program.

The final instruction, _delete_, enables the process to alter a data structure by removing branches from it. This has the effect of dissociating a substructure from the main data structure, and therefore preventing access to it to be gained from other parts of the data structure through use of _obtain node_. The deletion of a node can be accomplished by isolating it from its structure through _delete_, since the completion of program executions and the consequent destruction of activation maps, removes all means, i.e. pointers, for accessing the node.

Figure 3.7. shows an example of the use of these instructions. The instructions as they might appear in the programs are shown, as well as structure and process configurations at appropriate points in the execution. In the example, lower-case letters are used for variable names and integers for identifiers. The "standard" form of a newly initiated activation map is that variables a, b, and c are assigned pointer values for the program, root of the procedure structure, and root of the data structure, respectively.

An overall description of the actions specified by the example procedure is:

1) The process gains access to its associated table and the input source program, and creates a scratch segment called Temp.
2) It constructs an eds containing an intermediate representation of the program, and calls the directly associated procedure, supplying it with the newly created data structure.
3) The process alters the intermediate program representation while using a scratch segment of its own, which is deleted before control is returned to the main program.
4) The main program creates the object program, cleans up the data structure, and signals completion of its task.

(a) The program instructions in the order of execution, and indications of the points in the execution to which the following "snapshots" correspond.

```
     Instructions in Program 1          Instructions in Program 2
                                                                   <---- (b)
d = obtain node (b, Table)
e = obtain node (c, Source)
f = create node (c, Temp, SEGMENT)
                                                                   <---- (c)
g = create node (c, Data, CONNECTOR)
h = create node (g, Intcode, SEGMENT)
k = obtain node (b, Subr)
    call (k, g)
                                                                   <---- (d)
                                     m = obtain node (c, Intcode)
                                     n = create node (c, Temp, SEGMENT)
                                         delete (c, Temp)
                                                                   <---- (e)
                                         return
l = create node (c, Object, SEGMENT)
    delete (c, Data)
    delete (c, Temp)
    return
                                                                   <---- (f)
```

(b) The structures and process state as execution is begun. (Since the eps does not change, it is not included in succeeding illustrations.)

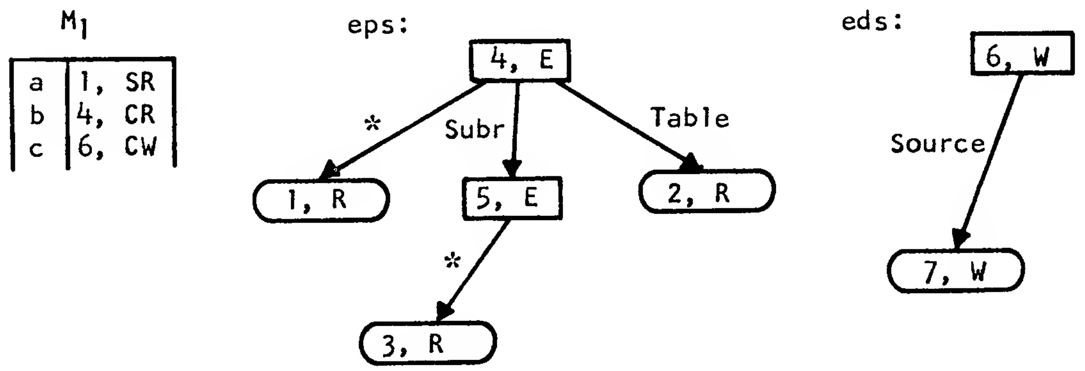
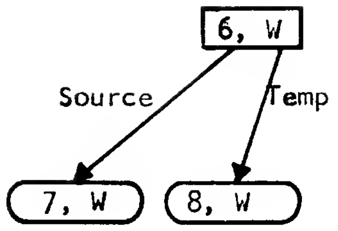


Figure 3.7. Example of Elementary Structure Use

(c)

M_1

a	1, SR
b	4, CR
c	6, CW
d	2, SR
e	7, SW
f	8, SW



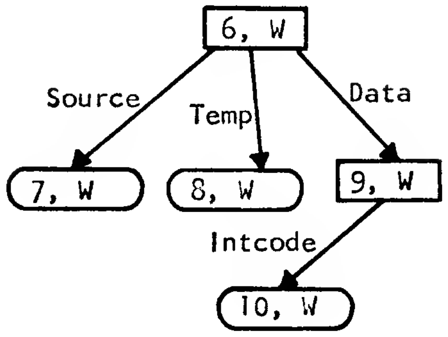
(d)

M_1

a	1, SR
b	4, CR
c	6, CW
d	2, SR
e	7, SW
f	8, SW
g	9, CW
h	10,SW
k	5, CE

M_2

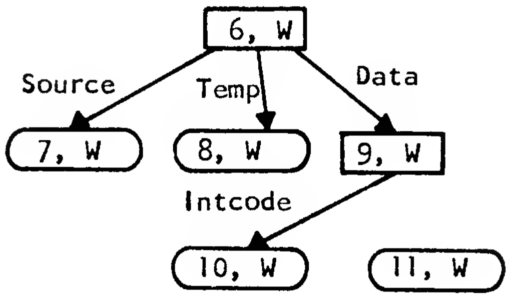
a	3, SR
b	5, CR
c	9, CW



(e)

a	1, SR
b	4, CR
c	6, CW
d	2, SR
e	7, SW
f	8, SW
g	9, CW
h	10,SW
k	5, CE

a	3, SR
b	5, CR
c	9, CW
m	10,SW
n	11,SW



(f)

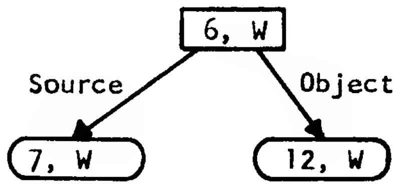


Figure 3.7. (continued)

3.5. Use of Elementary Structures by Processes

In this section, we demonstrate that the utilization of elementary structures by a process causes the structures to remain well defined. We also relate properties of the activities of a process, while using elementary structures, to the requirements given in Chapter 2 concerning the use of information by a process.

First let us define a property of elementary procedure structures which corresponds to a relationship discussed in Chapter 2.

Definition 3.6. Let P be an elementary procedure structure. A segment is *directly associated* with the program of P if there exists a branch from the root node of P to the segment. An information structure x is *directly associated* with the program of P if there exists a branch from the root node of P to the root node of x.

We show later in this section that this relation corresponds to the relationship discussed in Chapter 2 in terms of the requirements it places on the utilization of information.

Let us now consider the relationship between pointer types and node types, and their significance in determining how nodes may be used. In the case of pointers to connectors, the possible pointer types are CW, CR, and CE. From the descriptions of Table 3.1., we see that a type CW pointer must point to a type W connector which must be part of an elementary data structure. The instructions *obtain node*, *create node*, and *delete*, enable a process to make use of a type CW pointer for accessing nodes reached by branches leaving the node denoted by the pointer, building substructures connected to that node, and deleting existing substructures. A type CW pointer is also used by the instruction call to specify the root node of the data structure to be made available to the process during its execution of the called procedure.

The pointer types CR and CE both appear only in pointers to type E connectors, which occur only as root nodes of elementary procedure structures. Whether an activation map contains a type CR or CE pointer for the root node of an eps depends on whether the program of the procedure structure is being executed or the eps is directly associated with the program being executed. This is implemented in the following way. If the pointer is the root of the currently activated procedure, it must have been associated with the variable name x_1 by call and must be of type CR. A type CE pointer can be acquired only through use of obtain node starting at a node for which the process has a type CR pointer, which must be the root node of the currently activated procedure.

If a pointer specifies a segment, it must be of type SR or SW. This is because elementary information structures contain only segments of type R or W and obtain node, create node, and call cause the type of the pointer to reflect the segment type. There is no need for type E segments, since we are concerned with "executing" procedures, not programs. When a program segment is being used by a process, information from it is accessed as it would be from any other segment. The difference lies in the way in which the processor interprets the information. We assume that a type SR pointer enables the process to access, but not alter, the contents of the segment, and a type SW pointer allows both reading and alteration.

From this discussion, we can draw the following conclusions:

Proposition 3e.1.* A process does not alter an elementary procedure structure as it executes its program.

Proof: A process is initiated with a type CR pointer for the eps root and a type SR pointer for the program. Since the eps has only type R and E nodes, obtain node can yield only type SR or CE pointers. Also, call yields only type SR and CR pointers for the eps. Create node and delete can therefore never be used in conjunction with an elementary procedure structure node, since they require type CW pointers. Also only type SR pointers can be obtained for segments, and thus no segment of a gps can be altered.

Proposition 3e.2. The actions of a process must result in the elementary data structure supplied to it being transformed into another elementary data structure.

Proof: Changing the contents of a segment will not affect the "form" of an eds. Thus only the two operations which cause structure alteration,

---

* The e in 3e.1. indicates that the proposition concerns elementary structures. Later in this chapter, we consider properties of general structures.

create node and delete, must be shown to transform an eds into another eds. Create node behaves properly since it creates a branch to a newly created node, which must therefore have no other branches terminating on it. The instruction delete also causes no problem since it can only break off portions of the eds. At least the root node of the original eds must remain. Since this is a well defined eds, use of delete always causes an eds to remain.

Proposition 3e.3. If P is an eps, the only segments of P, which may be referenced by a process executing the program segment of P, are those which are directly associated with that program.

Proof: Assume the process is using M_k. By definition of an eps, only type R and E nodes may exist in P. The instruction which allows other nodes of P to be accessed is obtain node. Since the pointer $f_k(x_1)$ to the root node of P is of type CR, obtain node can be used to access each of the nodes reached by branches leaving P's root node. Each of these nodes must be either a type R segment or a type E connector. In the latter case, the new pointer is of type CE. Obtain node cannot be used to traverse a branch leaving a node for which a process has only a type CE pointer. Thus, no segments may be accessed by paths through the type E connectors, and the only accessible segments are those reached directly from the root node of P.

Proposition 3e.4. If a process is using activation map M_k, it has the ability to access each of the nodes contained in the eds whose root node is pointed to by $f_k(x_2)$. Furthermore, the structure may be changed into any desired form. (The form of a data structure can be considered to be a skeleton of the structure which ignores the node identifiers and the segment contents, but retains the node classes and types and the named branches relating them.)

Proof: The first part follows directly from the description of obtain node, since any node can be accessed by its repeated use. The second follows from the ability of delete to remove arbitrary portions of the structure and repeated use of create node to form arbitrary substructures.

The second type of property of the model which is of interest concerns the means used to name information. Through use of the mechanisms just described, processes can identify information from the storage subsystem without having any global, i.e. unique, identification, such as a pointer value, stored within the program being executed. Identification of and access to information is determined by the value of a variable. This variable value exists either from being predefined for the program activation, or by causing its definition through one of the described instructions. But information is specified to these operations relative to some already defined variable through the use of a branch name. Thus the only naming information needed within a program are variable names and branch names used in the structures being referenced.

The following consequence can be deduced from this property:

Proposition 3e.5. A process is able to access only information which is part of one of the structures to which it was given root node pointers when initiated.

Proof: This follows immediately from the relative naming methods used for the program instructions.

The first two propositions indicate that the execution of the program of an eps must leave the structures used by the process well defined. The last three propositions relate to the requirements of Section 2.5. If we restate Requirements 5 and 6 in terms of elementary structures and processes using them, we obtain the following two theorems:

_Theorem 5e._* Let P be an eps. During execution of the program of P, a process can access the data segments directly associated with that program, the nodes of the eds supplied for that activation by the initiator, and the nodes created during the program execution. Furthermore, the process may create other eds nodes and utilize them.

Proof: Assume the process is executing the program using map M_k. Proposition 3e.3. indicates that the directly associated data can be accessed. The supplied data and that created by the process during execution of the program must be part of the eds pointer to by $f_k(x_2)$, and thus are accessible by Proposition 3e.4. Since $f_k(x_2)$ is a type CW pointer, the process can attach new nodes, by use of _create_ _node_,

* Here 5e indicates Requirement 5 of Section 2.5. related to elementary structures.

to the connector denoted by the pointer at any time during the program execution.

Theorem 6e. Let P be an eps. During execution of the program of P, a process can request the execution of any eps directly associated with the program of P. However, while executing the program of P, the process cannot access the program of any directly associated eps or any of the segments or structures directly associated with that program (unless they are also directly associated with the program of P).

Proof: Assume the program of P is being executed using map M_k. Through the use of x_j obtain node (x_1, b) with appropriately chosen branch name b, the value $f_k(x_j)$ can be made a type CE pointer to the root node of any directly associated eps. The use of the eps could then be requested using call (x_j, x_k), where x_k is any type CW pointer (pointing to a connector of the eds being used). Proposition 3e.3. tells us that no pointers for other nodes of the directly associated eps can be obtained, while the process is using M_k, unless the nodes are also directly associated with the program of P.

An additional property of elementary structures, which we have not yet considered, relates to the possibility of name ambiguities. As we indicated in Chapter 2, existing systems, which provide only one directory for the information utilized by each principal, require that each file have a unique name. Thus, the names, for other information, in programs (both owned and borrowed) used by the principal must not conflict. Use of a tree-like directory structure, as in the MULTICS system, allows the same entry name to be used for more than one file. However this required some additional steps to be taken so that, when a program refers to a file by an entry name, the intended file is utilized.

This possible ambiguity of file names, however, can be avoided by the use of elementary structures. As Proposition 3e.5. indicates, a process can make use only of nodes contained in the structures, whose root nodes are pointed to by $f_1(x_1)$ and $f_1(x_2)$. Those nodes are accessed through the use of instructions, in which each node is identified (uniquely) by the name of a branch leaving a node which has already been accessed.

To see how this mechanism can be used to avoid name ambiguities, consider the example in Figure 3.7. The eps makes available to the process all information which is associated with the procedure program by the procedure creator. This information can be accessed by the process through the use of <u>obtain</u> <u>node</u>, since it has a pointer to the eps root node ($f_1(b)$ in the example) and "knows" the names of branches leaving that node. Similarly the process can access the information in the eds supplied by the initiator of the procedure execution. The other information needed by the process is that which is created by the process.

Both programs in Figure 3.7. cause a branch named Temp to be created leaving the node specified by the variable c. No ambiguity results, since the subprocedure is supplied with an eds containing only the information (node 10) it requires. The program of the subprocedure, "knowing" that it will be supplied with an eds such that the only branch leaving the eds root node has name "intcode", can create the additional branch without causing any name conflicts. Of course, if the main program had just attached node 10 to node 6 and allowed the subprocedure to access that node, ambiguity would have occurred. Thus, if each program activation is supplied with an eds of the appropriate form (in the sense described in Proposition 3a.4.), no ambiguity can result.

3.6. General Information Structures

The structures discussed in 3.1. and 3.2. are not adequate to model all types of stored information. We have shown no way to incorporate alterable data into a procedure, which we argued in Chapter 2 should be the means for maintaining shared data bases. Also, we have no way to incorporate read-only data into data structures or to pass procedures as arguments to subroutines. It is the role of this section to generalize the mechanisms discussed above to allow these capabilities, and to demonstrate that the properties of the structures are essentially unchanged.

First, we shall define a new type of structure, called a basic structure. This is simply an information structure composed of read-only data. The class is defined as follows:

Definition 3.7. The class of basic structures is the set of all information structures satisfying the following property:

> The root node of the structure is a type R connector with one or more A-branches leaving it, each with a different name, with each branch terminating on a segment of type R or the root node of a basic structure.

A basic structure will be abbreviated by bs.

In order to generalize the procedure structure and data structure, it is necessary to define them in terms of one another in a recursive fashion. This has the effect of allowing data structures to contain read-only data and procedures, and allowing procedure structures to contain alterable (data) structures as components.

Definition 3.8. The class of general data structures is the set of all information structures satisfying the following property:

The root node of the structure is a type W connector which has zero or more A-branches leaving it, each with a different name, and each terminating on any of the following:

1) a segment of type W or R,

2) the root node of a basic structure,

3) the root node of a general procedure structure, or

4) the root node of a general data structure.

Furthermore, each type W node of the structure has no more than one branch of the structure terminating on it.

Definition 3.9. The class of general procedure structures is the set of all information structures satisfying the following property:

The root node of the structure is a type E connector with one *-branch leaving it which terminates on a type R segment, and zero or more A-branches, each with a different name, and each terminating on any of the following:

1) a type R segment,

2) the root node of a basic structure,

3) the root node of a general data structure, or

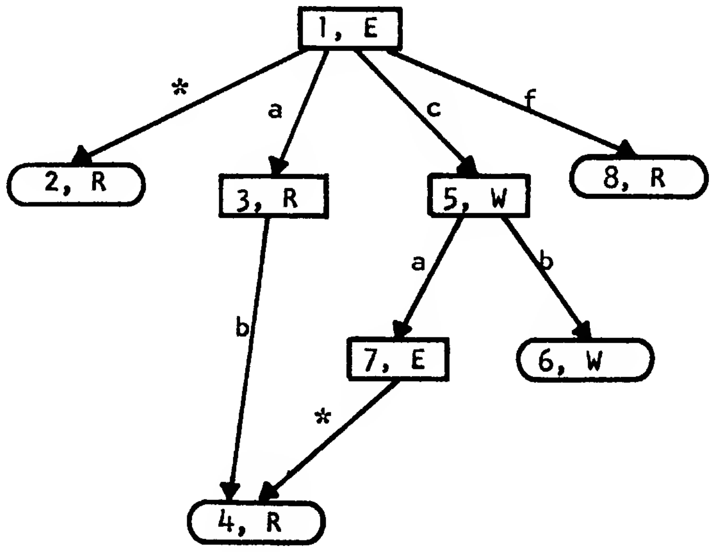
4) the root node of a general procedure structure.

These two new types of structure are abbreviated as gds and gps. Examples of these structures are shown in Figure 3.8. We should observe that the elementary structures defined earlier are special cases of these general structures.

Basically the general procedure structure allows alterable information to be associated with a procedure. Thus a procedure may retain information generated by its past activations, and, for example, may maintain an alterable data base. The general data structure allows read-only data as well as alterable data to be associated with a data structure, and to be passed to a procedure as parameters of a call. In addition, a general data structure may contain procedures, that is general procedure structures, so that they also might be supplied to a procedure for a particular activation.

The same form of activation map and set of instructions can be utilized for modelling activations of general structures as for elementary structures with one minor modification. A mechanism is required to keep alterable data associated with a procedure (Category I information) separated from the (Category II) information associated only with that activation. This separation ensures that the data structure supplied to each procedure activation, is a substructure of the gds originally supplied to the process, and thus is owned by the process owner. The importance of this will be seen in Chapter 4.

(a) General Procedure Structure



(b) General Data Structure

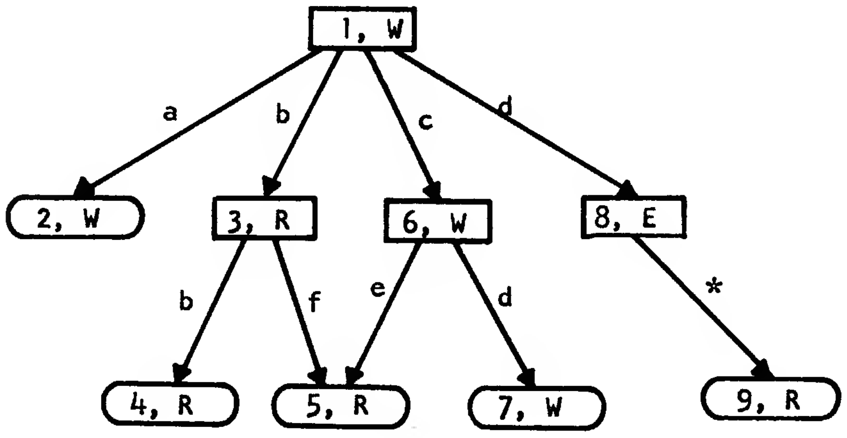


Figure 3.8. Examples of General Structures

The implementation of this mechanism is relatively straightforward. Each type CW pointer is classified as internal or external, which is intended to indicate whether the connector is part of the procedure structure or not. When a process is initiated, the pointer $f_1(x_2)$ is made to be external. This indication is propagated by $x_j =$ obtain node (x_i, b) and $x_j =$ create node $(x_i, b, \left\{ \begin{array}{l} \text{segment} \\ \text{connector} \end{array} \right\})$ by causing $f_k(x_j)$ to be external just if $f_k(x_i)$ is. When call (x_i, x_j) is executed, the pointer $f_k(x_j)$ is required to be external, and $f_{k\,1}(x_2)$ also becomes external. Use of these mechanisms thus ensures that the general data structure passed to a procedure activation is a substructure of the data structure originally supplied to the process.

3.7. Use of General Structures by Processes

This section contains a description of properties of a process' use of general structures analogous to those discussed in 3.5. for elementary structures.

First we define direct association for general procedure structures. To do this, we need the notion of a path in an information structure, and of particular types of paths.

Definition 3.10. A path in an information structure s is a sequence of A-branches $x_1, x_2, \ldots, x_j$ in s, such that x_1 leaves the root node of s, and each x_i, for $i>1$, leaves the node on which x_{i-1} terminates. The path name of the path is $b_1.b_2. \ldots .b_j$, where b_i is the branch name of x_i.

Definition 3.11. A path of type W in an information structure s is a path in s for which each node from which a branch of the path emanates, with the possible exception of the root node of s, is of type W. That is, all branches of the path, except perhaps the one leaving the root node of s, leave nodes of type W. A path of type W-R in s is a path in s for which each node from which a branch of the path emanates, with the possible exception of the root node of s, is of type W or type R.

As we shall see, these are the paths which can be traversed using obtain node, and thus nodes which can be reached by such paths are "accessible" to the process if it has a pointer to the root node of the structure. The exception of the structure's root node allows us to describe such paths in both procedure and data structures.

Definition 3.12. Let P be a general procedure structure. A segment is *directly associated* with the program of P if there exists a path in P of type W-R to the segment. An information structure x is *directly associated* with the program of P if there exists a path in P of type W-R to the root node of x.

This definition causes all of the data, both type R and W, contained in P which can be reached without passing through any type E connectors to be directly associated with the program of P.

Now let us turn to the properties of the mechanisms for utilizing general structures. The propositions are numbered to correspond to those given for elementary structures in Section 3.5.

Proposition 3g.1. The actions of a process in executing the program of a general procedure structure always result in the gps being transformed into another gps.

Proof: The difference between this case and that of elementary procedure structures is that it is now possible for the process to alter the data structures which are part of the procedure structure. The process, using *obtain node*, can obtain type CW pointers for the root nodes of the gds's reached by branches from the procedure structure's root node. The question then is whether these data structures are always transformed into other gds's by the execution of program instructions. Again, create node and delete are the only instructions which can alter the form of a gds. Create node can establish branches only to new nodes, which therefore have no other branches terminating on them, and delete can only remove substructures of the gds. Thus a gds must remain after use of these operations, and thus the procedure structure will still be a general procedure structure.

Proposition 3g.2. The actions of a process always result in the gds supplied to it being transformed into another well defined gds.
Proof: This result is shown by the above proof for Proposition 3g.1.

Proposition 3g.3. Let P be a gps. The only nodes of P which may be accessed by a process while executing the program of P are those segments which are directly associated with the program of P, and the root nodes of structures directly associated with the program of P.
Proof: A process may "traverse" a path in P to a node through the use of obtain node. At each step, obtain node requires that the process have a type CR or type CW pointer for the connector from which the branch to be traversed leaves. But the process could have this pointer only if it were traversing a path of type W-R in P.

A similar argument can be made for nodes accessed by create node. Thus, by Definition 3.12., these segments and root nodes must be directly associated with the program of P.

Proposition 3g.4. If a process is using activation map M_k, it has the ability to access each of the nodes reached by a path of type W-R in the gds whose root node is pointed to by $f_k(x_2)$. Furthermore, the process may alter the type W paths of the gds and any segments reached by such paths.

Proof: The first part is true by the same argument used for Proposition 3g.3. The second part is true because the process may obtain a type CW or SW pointer for a type W connector or segment reached by a path of type W, through the use of obtain node. Thus, if the node is a connector, the process may add or delete branches leaving the node. If it is a segment, the process may alter the contents of the node.

This proposition does point up one dissimilarity between the use of general and elementary structures. A process can create an eds of arbitrary form to pass to a subprocedure. However, since a process has no means to create a gps or bs, it cannot construct a gds of arbitrary form, but may only add type W nodes. This is a slight constraint on the forms of information which can be passed between general procedures, but is necessary to attain the controlled sharing discussed in Chapter 4. Chapter 6 contains a discussion of this constraint and a means for overcoming it.

Proposition 3g.5. A process can access only information which is part of one of the structures to which it was given a root node pointer when initiated.

Proof: This is the same as Proposition 3e.5. and is true for the same reasons.

The first two propositions tell us that the execution of the program of a gps must leave the structures used by the process well defined. The last three relate to the requirements of Chapter 2, as indicated by the following theorems:

Theorem 5g. Let P be a gps. During execution of the program of P, a process can access all segments directly associated with that program, all root nodes of directly associated general data structures and basic structures, and all type R or W nodes reached by paths of type W-R in the gds supplied for that execution. Furthermore, the process may create other gds nodes and utilize them.

Proof: Propositions 5g.3. and 5g.4. indicate that this is true.

Theorem 6g. Let P be a gps and Q be the gds supplied for the execution of P. During the execution of the program of P, a process can request the execution of any gps directly associated with that program or any gps, contained in Q, whose root node can be reached by a path of type W. However, while executing the program of P, the process cannot access the program of such a gps, or any of the segments or structures contained in such a gps, unless they can also be reached by paths of type W-R in P or Q.

<u>Proof:</u> From Propositions 3g.3. and 3g.4., the process can access the root nodes of these procedure structures. Since <u>call</u> requires only a pointer to the gps' root node, this is adequate to request its execution. The same propositions tell us that nodes may be accessed only by traversing paths of type W-R. Thus, nodes accessible only by paths through type E connectors may not be accessed by the process while executing the program of P.

CHAPTER 4

ENVIRONMENTS

4.0. Introduction

The last chapter introduced some forms of structured information. In addition, it showed how means can be provided, for a process to utilize information, which satisfy the requirements of controlled information sharing. In this chapter, we show how information can be associated with and used by principals to accomplish controlled sharing of information.

A means for organizing stored information is first described. Analogous to the use of structures by processes, the form of this organization determines what information each principal may use and constrains the type of use he may make of it. A set of commands is also specified, which defines the types of activity which a principal may request the utility to perform. These enable a principal to control procedure executions, form new procedures, and share procedures with other principals. Arguments are then given to show that properties of the model can be directly related to the requirements of controlled information sharing discussed in Section 2.5.

We should note again that the model described here allows a relatively unrestricted type of information sharing; a principal is allowed to share any owned procedure. Chapter 5 contains a discussion of other sharing disciplines and the model modifications required to implement them.

Analogous to Chapter 3, the development in this chapter is separated into two parts. First a model is described which makes use of elementary information structures. Then the model is altered to allow use of general structures.

4.1. Elementary Environment

In this section, a model of a means for organizing stored elementary information structures is defined, and mechanisms which enable principals to make use of the information is described.

Structure of the Elementary Environment

The means by which information is organized and by which access abilities are associated with principals is the elementary environment. The elementary environment is a collection of information structures, each one being associated with a particular principal. Each of these structures is referred to as an elementary principal structure, abbreviated eprs. The form of a principal's eprs exhibits which information is accessible to him, and thus it determines which stored information he can utilize and how he can use it. Sharing is accomplished by providing a means for information structures to be contained in the eprs's belonging to different principals.

All information in the elementary environment is owned by some principal, who has control over how it is used. The ownership of information private to a principal, e.g. data structures, is specified by the form of the environment. An additional indication is required to determine the owner of shareable information, however, and this is provided by a function called the ownership function.

<u>Definition 4.1.</u> An <u>elementary environment</u> consists of three components:

1) A set of principals $P = (p_1, p_2, \ldots, p_K)$

2) A set of information structures $F=(f_1, f_2, \ldots, f_K)$ in which

   a) the root node of each f_i is a type W connector and is not contained in any of the other structures of F (that is, there is no other structure f_j with a path in f_j to the root node of f_i), and

   b) the root node of each f_i has zero or more A-branches leaving it, each with a different name, such that each branch terminates on the root node of an elementary procedure structure or an elementary data structure, and

   c) any eds which is directly attached to f_i has only one branch terminating on its root node, and is disjoint from any other eds which is directly attached to any $f_j (j=1, 2, \ldots K)$. (An eds is said to be directly attached to f_i if there exists a branch from the root node to f_i to the root node of the eds).

3) A function, O, which maps each type E connector of each element of F into an element of P.

Each element of F is called an elementary principal structure, abbreviated eprs.

Relationships between Principals and Information Two relationships between a principal and information determine his ability to use that information: ownership and direct (and indirect) accessibility. Both can be determined by the form of the elementary environment in conjunction with the function 0.

As we discussed in Chapter 2, only procedures may be shared. Thus data structures which are substructures of an eprs should be owned by the associated principal. The ownership relation for structures is determined by the following definition. (In this discussion, the types of paths defined in the last chapter are used.)

<u>Definition 4.2. (Part 1)</u> Each principal structure f_i is <u>owned</u> by principal p_i. Each elementary data structure whose root node is reached by a path of type W in f_i is <u>owned</u> by p_i. An elementary procedure structure is <u>owned</u> by the principal which 0 associates with its root node.

This definition associates exactly one owner with each eps of F, since 0 associates a unique principal with each type E connector. Consider now a type W connector which is not the root node of some f_i. It can be part of only one directly attached eds, since the directly attached eds's are disjoint. Each directly attached eds has only one branch to its root node, and thus the type W connector must be part of one eprs, say f_i. The eds, of which this connector is the root node, therefore is owned only by p_i, and each type W connector has exactly one owner.

Since the ownership of a structure is determined by a condition regarding its root node, and each connector is the root node of a structure, the ownership of a structure can be thought of as synonomous with the ownership of its root connector. Thus we can define the notion of an owned path.

<u>Definition 4.3.</u> An <u>owned</u> <u>path</u> in an elementary principal structure f_i is a path in f_i for which each connector from which a branch of the path emanates is owned by p_i.

We would now like to define ownership of segments by stating that p_i owns a segment if there exists an owned path in f_i terminating on it. This would be adequate in the case of type W segments, but if the segment is type R, it could have more than one branch terminating on it, each being part of an owned path in a different eprs. To resolve this problem, we need the idea of a well formed elementary environment (abbreviated wfee).

<u>Definition 4.4.</u> A <u>well</u> <u>formed</u> elementary environment is an elementary environment in which, for each type R segment,

1) all of the type E connectors, from which a branch leaves which terminates on the segment, are owned by the same principal, and
2) if the segment is reached by a path of type W in f_i, all of the type E connectors, from which a branch leaves which terminates on the segment, are owned by p_i.

This definition ensures that, if there are any owned paths to a type R segment, all of those paths are owned by the same principal. Ownership of segments can be specified by the following definition.

Definition 4.2. (Part 2) In a wfee, each segment reached by an owned path in f_i is owned by p_i.

This definition associates no more than one owner with each segment. If the segment is part of an eds, it is owned by the owner of the eds, as are all eps's making use of the segment, since the environment is well formed. If the segment is not part of an eds, it must be the program of, or directly associated with the program of, one or more eps's. Since each of these must be owned by the same principal, that principal is the only one who may own the segment.

Note that it is possible to have an "unowned" type R segment, since there may exist no owned paths to the segment. This means that the original creator of the segment has "cut it loose" by deleting all owned paths to it, but that use is still being made of procedures with which it is associated.

In the subsequent discussion, all elementary environments are assumed to be well formed. The idea of an owned path can be used to define direct accessibility.

Definition 4.5. An information structure x is directly accessible in an elementary principal structure f_i if there exists an owned path in f_i to the root node of x. A segment is directly accessible in f_i if there exists an owned path to it.

From Definition 4.2. each path of type W is an owned path, and thus all owned data structures and segments are directly accessible. Also, since paths of a single branch are owned, each procedure attached to the root node of an eprs is directly accessible. Finally, if a procedure is owned, the path to its subprocedures is owned, and each of those is directly accessible. This aspect of direct accessibility allows a principal to access the substructures of a procedure he created, without having to retain direct connections from his eprs root to each of the subprocedures. An example of a well formed elementary environment is shown in Figure 4.1.

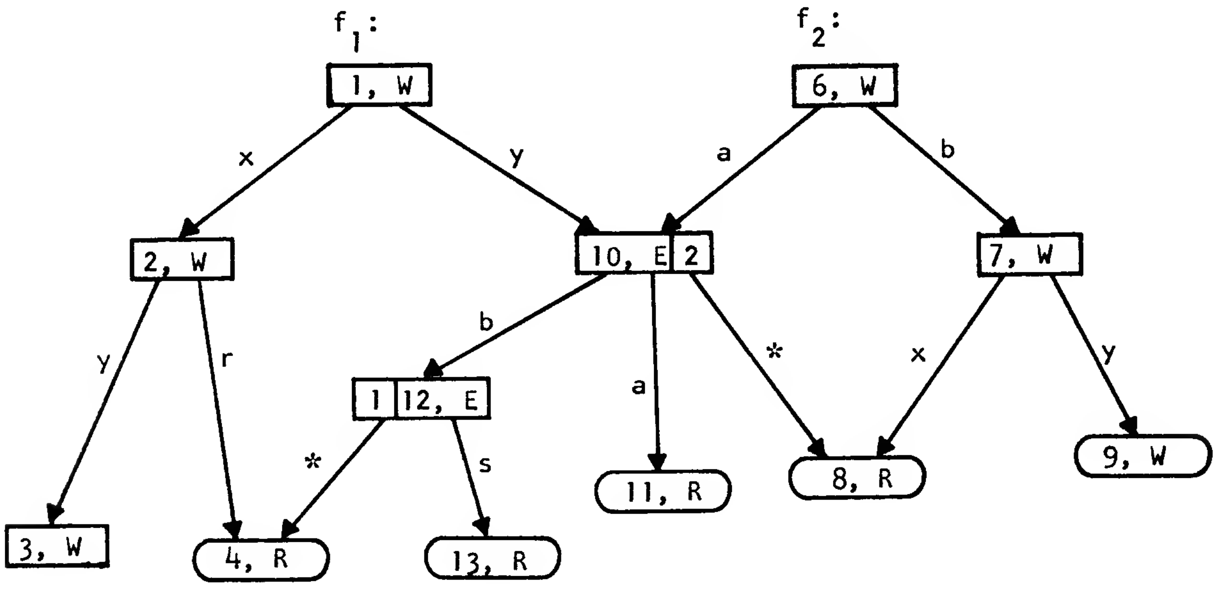


Figure 4.1. A Well Formed Elementary Environment

Connectors are shown as rectangles; segments as ovals. The components shown in each node are the node identifier and type, as in Chapter 3. In addition, a box is appended to each type E connector indicating the owner of the node; that is, the value of the function O. The figure shows two elementary principal structures. The eprs owned by p_1 contains a data structure (nodes 2, 3, 4) and a directly accessible procedure structure owned by p_2. The structure owned by p_2 also contains the procedure structure as well as a data structure (nodes 7, 8, 9) which contains (node 8) the program segment of the procedure structure. Note that the path named a.a from node 6 is an owned path, therefore node 11 is owned by p_2. However, node 13 is unowned since none of the paths to it are owned.

Instructions and Commands

To see how these structures can be formed and used, we shall describe the operations which can be performed on the structures. The operations are of two types: commands and instructions. Commands are directly requested by the principal, and specify actions to be performed utilizing directly accessible structures. Instructions are those actions discussed in Chapter 3 which can be requested by a process operating on behalf of a principal.

The commands are of three types. The first type requests the initiation or termination of a process, the second causes procedures to be formed, and the last implements sharing. The commands are described in Table 4.1. There we use the abbreviation "path name n denotes structure s" to mean that there exists an owned path in f_i with path name n which terminates on the root node of the structure s. Similarly "path name n denotes segment x" means that there exists an owned path with path name n which terminates on x. From the definition of direct accessibility, we can see that, since the paths must be owned, only directly accessible structures and segments may be "denoted".

The commands for controlling a process are quite straightforward. As part of the initiate process command, the principal specifies the root nodes of the eps and eds, to be used by the process, by path names. The command causes a process to be established with a state consisting of the single map M_1, initiated with values for x_0, x_1, and x_2. The principal may use this command only when no other processes owned by him exist. This restriction eliminates the need, in this basic model, for the additional complexity required for parallel processing. (Chapter 7 discusses a model generalization to allow parallel processing.) It is necessary that a principal be able to supply a process with the entire structure f_i as data, since a process must have access to the root node of f_i in order to create data structures contained by it. This proves to be no problem relative to the instructions executed by the process, since an eprs is clearly a special case of a gds, and we have seen that the instructions utilize generalized data structures in an appropriate manner. The command form eps seems to be quite complex, but its operation is straightforward as is shown by the example in Figure 4.2. (Only the directly affected portions of the environment

TABLE 4.1.

Assume these are issued by principal p_i.

1) Initiate process $(m, \{\underset{-}{n}\})$ - If path name m denotes an eps and path name n denotes an eds, and if there exist no processes owned by p_i, a process owned by p_i is initiated utilizing the designated structures. If n is not specified, the structure f_i is used in place of an eds.

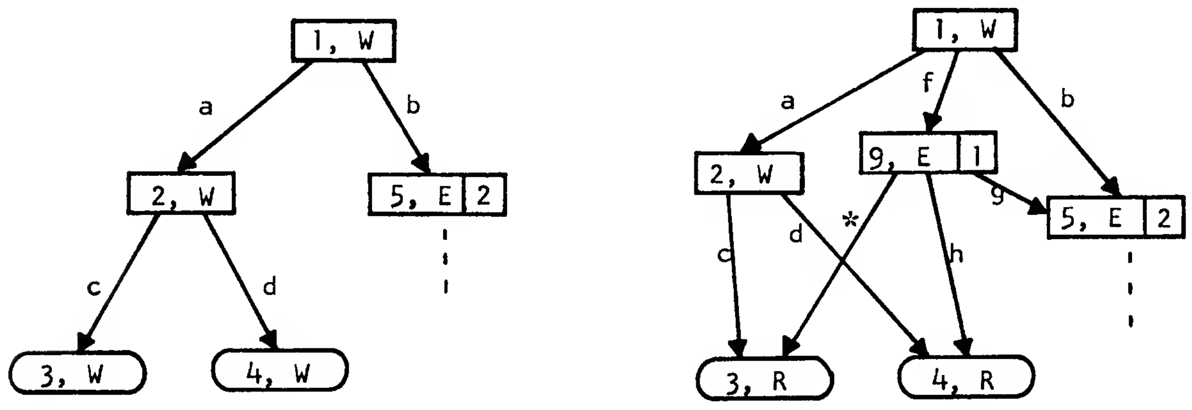
2) Terminate process - The process owned by p_i is destroyed.

3) Form eps b from $(n_0; \langle n_1, b_1 \rangle; \ldots; \langle n_J, b_J \rangle)$ - If b is a branch name not already occurring for a branch leaving the root node of f_i, n_0 denotes a segment, $n_1 \ldots n_J$ denote segments or eps's, and $b_1 \ldots b_J$ are distinct branch names, an eps is formed utilizing the segment denoted by n_0 as program segment and having branches from its root node with names $b_1 \ldots b_J$ terminating on the nodes specified by $n_1 \ldots n_J$. All of the denoted segments are made type R. A branch with name b is also constructed from the root node of f_i to the newly formed node. Principal p_i becomes the owner of the newly formed node.

4) Share n with S - If n denotes an owned eps, the set of principals S becomes the share set for the eps.

5) Borrow n from j as b - If p_i appears in the share set of the eps denoted in elementary principal structure f_j by n, then a branch with name b is constructed from the root node of f_i to the root node of the eps.

are shown.) Use of the command results in a new eps directly accessible, and owned by, p_1 which makes use of owned program and data segments (nodes 3 and 4) and a borrowed subroutine (node 5). Since the segments (3 and 4) are now part of an elementary procedure structure they can no longer be altered and thus are changed to type R.



form eps f from (a.c; ⟨a.d, h⟩; ⟨b, g⟩)

Figure 4.2. The form eps Command

The two commands, share and borrow, allow procedures to be shared among principals. The owner of the eps associates a share set, which is a set of principal names, with the procedure. Each principal whose name appears in the set then may establish access to it via a branch to its root node. It should be noted that this mechanism allows a principal to share any owned procedure, even though it may utilize non-owned subprocedures, which is the "least restricted" type of sharing discussed in Chapter 2. In the next chapter we describe how the model can be modified to allow other types of sharing involving greater amounts of control.

4.2. Properties of the Elementary Environment

Two types of properties must be demonstrated regarding the elementary environment. The first shows that the environment remains well formed and well defined. That is, a structure formed by alterations to a well formed elementary environment, resulting from the use of commands by principals, still satisfies the definition of a wfee. The second set of properties shows that the system organization, consisting of an elementary environment, satisfies the requirements described in Chapter 2.

Preserving Well Formedness

To demonstrate that use of the commands leaves a wfee well defined and well formed, assume that a well formed elementary environment, i.e. sets P and F and function O, already exists, and consider a particular eprs f_i. The following properties hold.

Proposition 4e.1. Use of commands by any principals cannot cause branches to be formed which terminate on the root node of f_i.

Proof: The only commands which can cause branches to be formed are form eps, borrow, and initiate process. Form eps forms only branches to segments or eps root nodes (type E connectors) and borrow forms only branches to eps root nodes, so neither of these could be used to form a branch to an eprs root node, which is a type W connector. A pointer to the root node of f_i could be given to a process owned by p_i, through his use of initiate process (b, -). However, since the process may only form branches to new nodes, this could not cause the creation of a branch to that node.

Proposition 4e.2. Use of commands by principals always results in f_i being transformed into another, well defined eprs, with its elementary data structures being disjoint from those of any other eprs.

Proof: First let us show the following Lemma.

Lemma 4.1. Use of commands by p_j cannot result in the alteration of f_i, if $i \neq j$.

Proof: The three commands which can result in alteration of the structure of F are initiate process, form eps, and borrow. Consider use of these by p_j, where $j \neq i$. Use of initiate process by p_j causes a process to execute the program of an eps using a supplied eds, which can be altered by the process. The eds is specified by a path name which, if p_j issues the command, is the name of a path in f_j to the root node of the eds. Since this eds is part of a directly attached eds in f_j, it is disjoint from any eds in f_i. Thus this eds may only have parts in common with eps's in f_i. However, the only parts they could have in command are type R segments, which may not be altered by the process. The use of form eps by p_j causes a new node to be formed, as well as branches to and from it, and some type W segments are changed to type R. The new node and branches are only on paths from the root node of f_j, and thus are not part of f_i. The type W segments must be part of eds's in f_j and also are not in f_i. Use of borrow by p_j causes only a branch to be formed from the

To see that the environment remains well formed, consider a type R segment x in F. Only the use of <u>form</u> <u>eps</u> can cause a branch terminating on the segment to be formed. If a principal p_i is able to construct a branch to x using <u>form</u> <u>eps</u>, there must already exist an owned path in f_i to x, and, since the environment is well formed, all owned paths to x must be in f_i. Use of <u>form</u> <u>eps</u> by p_i, however, only causes the creation of another owned path to x in f_i, since p_i owns the newly formed type E connector. Thus, no other principal may cause an owned path to x to be formed, and wfee's must be transformed into wfee's.

<u>Controlled Information Sharing Requirements</u>

Now let us consider the correspondence between the requirements for controlled sharing, restricted to cases modelled by wfee's, and properties of the well formed elementary environment. In the following discussion, the requirements of Section 2.5. are restated in terms of structures and relations defined for the wfee, and an informal proof of the validity of each is given.

<u>Theorem le.</u> Only elementary procedure structures can be directly accessible to more than one principal in a wfee.

<u>Proof:</u> We must show that the other forms of information, namely elementary data structures and segments, can be directly accessible to only one principal. If an eds is directly accessible to p_i then it is contained in a directly attached eds in f_i and is disjoint from any

of the eds's in f_j, where $j \neq i$. Thus its root node cannot be reached by a path of type W in f_j and is not directly accessible to p_j. By comparison of Definitions 4.3b. and 4.6., segments are directly accessible to p_i only if they are owned by p_i. Thus, since we have shown that each segment can have only one owner, only that principal may have direct access to it.

Theorem 2e. A principal may request the execution of any directly accessible eps, and may supply, for use during that activation of the eps program, any directly accessible eds. During the eps execution, information can be created, but all of that information, which is not deleted during the execution, becomes directly accessible to the principal requesting the execution.

Proof: The first part follows from the form of _initiate_ _process_ and Definition 4.6. If p_i requested the eps execution, the eds used by the process is in f_i. Information can be created by the process through _create_ _node_, which attaches the new node, by a branch, to a type W connector. This type W connector must be part of the eds, or have been separated from it by the use of _delete_. When the process is terminated, this information remains as part of the eds in f_i, or, if it was separated from the eds, it is destroyed.

Theorem 3e. A principal can form an elementary procedure structure using any segments or eps's directly accessible to him.

Proof: This follows directly from the form of _form_ _eps_.

Theorem 4e. Use of form eps or initiate process by a principal cannot affect the directly accessible information of any other principal.

Proof: This follows directly from Lemma 4.1., which is shown in the Proof of Proposition 4e.2.

Theorem 5e. Let P be an eps. During execution of the program of P, a process can access the data segments directly associated with that program, the nodes of the eds supplied for that activation by the initiator, and the nodes created during the program execution. Furthermore, the process may create other eds nodes and utilize them.

Proof: This is proved in Section 3.5.

Theorem 6e. Let P be an eps. During execution of the program of P, a process can request the execution of any eps directly associated with the program of P. However, while executing the program of P, the process cannot access the program of any directly associated eps or any of the segments or structures directly associated with that program (unless they are also directly associated with the program of P).

Proof: This is proved in Section 3.5.

Theorem 7e. Each eps created by a principal becomes owned by him.

Proof: This follows directly from the description of form eps and Definition 4.3.

Theorem 8e. A principal can make the use of any owned eps available to any principal or group of principals.

Proof: This is directly implemented by the share command.

Theorem 9e. A principal may establish direct access to any eps made available to him.

Proof: Borrow causes a branch to be formed from the root node of the principal's eprs. Since any path of only one branch is an owned path, the eps is directly accessible.

This discussion has shown that the relations of ownership and direct accessibility defined in this chapter correspond to the relationships described in Chapter 2. Thus, if a computer utility were designed with properties representable by those of the well formed elementary environment, users could make use of the utility to share and utilize information in a controlled way.

4.3. Sharing Decisions by Algorithm in the Elementary Environment

As we indicated in Chapter 2, the use of a share set is just one method by which a potential borrower's rights may be determined. It would be desirable, in general, to allow the procedure owner to specify an algorithm which determines how sharing decisions for that procedure are to be made.

An implementation of this can be provided by extending the wfee of 4.1. The following is a typical scenario for this type of sharing. A principal, say p_i, owns elementary procedure structures $B_1 \ldots B_h$ which he desires to make available to others. He specifies the algorithm by which the suitability of potential borrowers would be judged regarding their use of $B_1 \ldots B_h$, and writes a program, say C, which implements the algorithm. Principal p_i then forms an eps A using C as its program and $B_1 \ldots B_h$ as subprocedures, and creates a share set for A which makes A available to potential borrowers (perhaps everyone) of $B_1 \ldots B_h$.

A principal p_j who desires to use one or more of the procedures $B_1 \ldots B_h$, uses <u>borrow</u> to obtain direct access to procedure A. He then initiates a process to execute procedure A and supplies it with an eds from f_j which contains, for example, a specification of which procedures are desired, along with a password or other identification information. The algorithm in C is executed and a decision is made regarding the procedure access which p_j is to obtain.

One piece of information which could be important to C is the identity of the potential borrower. The borrower could not be trusted necessarily to give his own name, so an instruction should be provided to supply this information to the process as it is executing C. This instruction could be of the form

u = <u>process</u> <u>owner</u> - If u is a non-pointer variable name, and p_i is the process owner, then u is assigned the value i.

It supplies the process with the identity of the process owner, which can be used as information for the algorithm specified by C.

After a decision to share a procedure has been made, a mechanism must be provided to implement the decision. Sharing of procedures can be accomplished by the process, while still executing C, through the use of a new instruction, <u>connect</u>. This instruction is defined in the following way, assuming that the process is using activation map M_k:

<u>connect</u> (x_i, b) - If $f_k(x_i)$ is a pointer to a type E connector, and if $O(f_k(x_1))$ (owner of the executing procedure) is the same as $O(f_k(x_i))$, and if a branch named b does not already leave the root node of the principal structure of the process owner, a branch named b is constructed from that node to the node pointed to by $f_k(x_i)$.

Use of the instruction causes a branch, with name b, to be formed from the root node of the borrower's eprs (the borrower being the process owner) to the root node of the eps being shared.

In order for <u>connect</u> to be executable, the owner of the sharing algorithm (and thus of the procedure containing it) must be the same as that of the procedure being shared. This ensures that this mechanism cannot be used by a principal to share direct access to a procedure he does not own.

Use of connect does alter the properties discussed at the end of Section 4.1. First, it alters the properties regarding process access abilities since the process may cause a branch to be attached to the root of the process owner's eprs, which may not be part of the data structure supplied to the process. However, since this node is accessible to the process owner only, the directly accessible information of other principals is not affected.

Secondly, of course, it provides an additional means for procedure sharing which cannot be used for every owned eps. That is, there must exist a procedure which a borrower accesses through borrow before connect can be used. However, that is no real restriction, since any eps can be made a substructure of another, whose program may use connect, so that use of any procedure may be shared through a sharing algorithm utilizing connect.

4.4. General Environment

In this section, we are interested in studying a more general form of environment allowing the inclusion of general procedure structures and general data structures. Use of these structures allows users to interact more, such as through shared, alterable data bases. However, mechanisms analogous to those discussed for elementary environments still provide control of sharing in the sense discussed in Chapter 2.

The general environment is defined in a similar manner to the elementary environment.

Definition 4.6. A general environment consists of three components:

1) A set of principals $P = (p_1, p_2, \ldots, p_k)$.
2) A set of information structures $G = (g_1, g_2, \ldots, g_k)$ in which
   a) each g_i is a general data structure,
   b) the root node of each g_i is not contained in any of the other structures of G, and
   c) each type W node in G, which is not the root node of a g_i, has exactly one branch terminating on it.
3) A function O mapping each type E connector of each element of F into an element of P.

Each element of G is called a general principal structure, abbreviated gprs.

Relationships between Principals and Information

The relationships of ownership and direct accessibility are defined similarly to the same relationships for the elementary environment. First we define ownership for general principal structures and some of the structures they contain.

Definition 4.7. (Part 1) Each g_i in G is owned by principal p_i. Each general data structure whose root node is reached by a path of type W in g_i is owned by p_i. Each general procedure structure is owned by the principal which the function O associates with its root node.

Clearly this definition associates exactly one owner with each g_i and each gps. Also, each type W node can have only one branch terminating on it. Thus, if there exists a path of type W in g_i to an eds root node, that must be the only path to that node, and p_i is the unique owner of the eds.

In order to define ownership for basic structures and segments, general environments must be restricted to be well formed. The type R connectors of basic structures cause a problem similar to that caused by type R segments in the elementary environment, and thus the ownership of paths to them must be constrained just as those to type R segments.

Definition 4.8. A well formed general environment is a general environment in which, for each type R node (connector or segment),

1) all type E connectors, from which a branch leaves which terminates on the node, are owned by the same principal, and
2) if the node is reached by a path of type W-R in a gprs g_i, all type W-R paths to the node are in g_i, and all type R or E connectors (excluding those type R connectors which are unowned), from which a branch leaves which terminates on the node, are owned by p_i.

This definition guarantees that all owned paths to type R segments and to basic structure root nodes are owned by the same principal. Ownership can now be extended to basic structures and segments.

Definition 4.7. (Part 2) A basic structure whose root node is reached by an owned path in g_i is owned by p_i. A segment reached by an owned path in g_i is owned by p_i.

Any eds or bs or segment reached by a path of type W-R in a structure g_i is owned by p_i. If a node can only be reached through a type E connector, it may or may not be owned. From Definition 4.7., if there exists an owned path in g_i to a gps root node and p_i owns the gps, p_i also owns the gps program segment, the basic structures directly connected to the gps root node, and the segments of the basic structures. However, neither p_i nor any other principal owns any general data structures accessible from the gps root node nor any of the type W segments they contain. This is consistent with the fact that any alterations to a gds associated with a procedure can be caused only through use of that procedure. All principals which may execute the procedure thus have equal capability to alter the gds, and thus no one can be said meaningfully to own it.

The direct accessibility relation can be defined as follows.

Definition 4.9. A general procedure structure or a basic structure is *directly accessible* in a gds g_i if there exists an owned path in g_i to the structure's root node. A general data structure is *directly accessible* in g_i if there exists a path of type W to its root node. A segment is *directly accessible* in g_i if there exists an owned path to it.

This definition excludes general data structures associated with procedures by requiring that the path be of type W, and thus not pass through the root node of a procedure. Since there is only one path to an associated type W connector from the procedure root node (by the definition of a gds), all paths to the connector must pass through the procedure's root node, and thus would not be of type W.

A well formed general environment is shown in Figure 4.3.

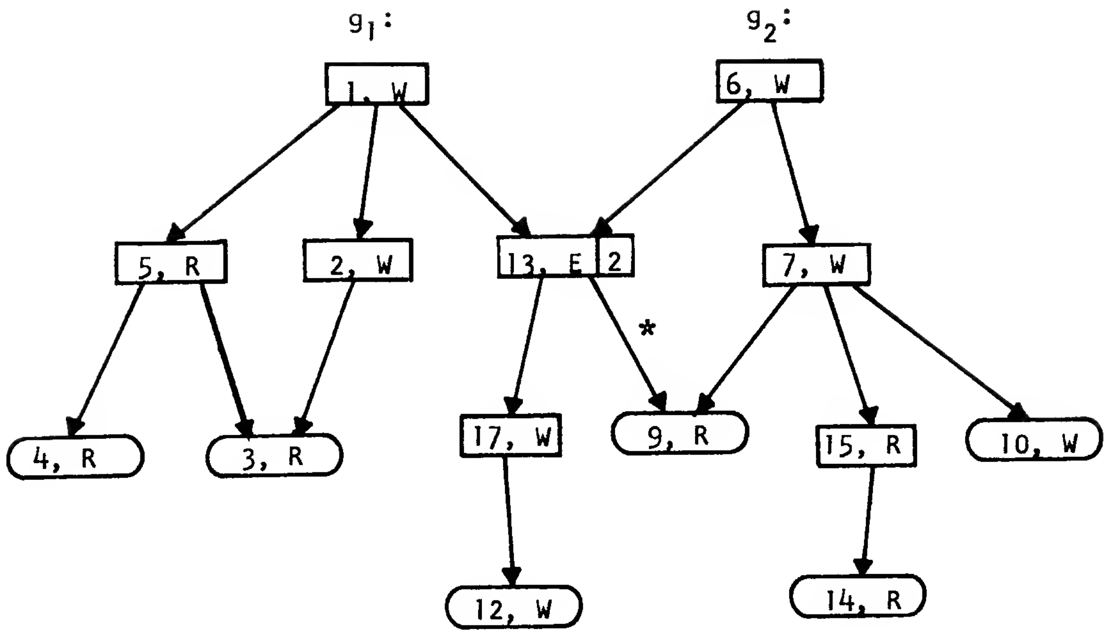


Figure 4.3. A General Environment

Principal p_i has direct access to a basic structure (root node 5), a general data structure (root node 2), and a general procedure structure not owned by him (root node 13). He also has direct access to segments 4 and 3. Principal p_2 has direct access to a gps (root node 13) which he owns, and a gds (root node 7) which contains a bs (root node 15). Note that the gds (nodes 11 and 12) contained in the procedure structure is not owned (or directly accessible) to either principal.

Instructions and Commands

The form of a process in a general environment and the instructions it may use are the same as those described in Section 3.6. The commands are analogous, with one addition, to those of Table 4.1., and are shown in Table 4.2. Again we use the phrase "the path name n denotes a structure (or a node)" to mean that there exists a path, with name n, in g_i which terminates on the root node of the structure (or on the node), and which ensures that the structure (or node) is _directly accessible_ to p_i. Thus, if a gps, bs, or segment is denoted, the path must be owned; if a gds is denoted, the path must be of type W.

The commands _initiate process_ and _terminate process_ are analogous to the same commands for elementary environments. _Form gps_ is similar to _form eps_, except that the gps may be attached to any directly accessible (and owned) gds root node, and that general data structures may be made part of a gps. The first difference only implies a change in the syntax of _form gps_ so that the name of the newly formed structure is specified as a path name followed by a branch name. It is required that any gds contained in a gps can no longer be directly accessible to any principal. Thus, each branch which connects the root node of a gds, incorporated into the new procedure, with g_i is deleted by use of _form gps_.

The command _form bs_ operates in a similar manner to _form gps_, except that only segments and other basic structures may be used as components, and that there is no program segment. Thus no deletion of existing branches is necessary. _Share_ and _borrow_ are also similar to those of Table 4.1., except that borrow can attach the newly accessed procedure to any directly accessible type W connector.

Note that the commands _form gps_ and _borrow_ allow the branches to

TABLE 4.2.

Assume these are issued by principal p_i.

1) Initiate process $(m, \left\{\begin{smallmatrix} n \\ - \end{smallmatrix}\right\})$ - If path name m denotes a gps and n a gds, and if there exist no processes owned by p_i, a process owned by p_i is initiated utilizing the designated structures. (If n is not specified, the structure g_i is used as the gds.)

2) Terminate process - The process owned by p_i is destroyed

3) Form gps $n{:}b$ from $(n_0; \langle n_1, b_1\rangle, \ldots, \langle n_J, b_J\rangle)$ - If n denotes a type W connector without a branch named b leaving it, n_0 denotes a segment, $n_1 \ldots n_J$ denote already existing nodes and $b_1 \ldots b_J$ are distinct branch names, a gps root node is formed using the segment denoted by n_0 as program segment and having branches from the new node, with names $b_1 \ldots b_J$, to the nodes specified by $n_1 \ldots n_J$. All of the denoted segments are made to be type R. For each n_i denoting a type W connector, the last branch in the path n_i is deleted. A branch with name b is constructed from the connector denoted by n to the newly formed node. Principal p_i is the owner of the newly formed node.

4) Form bs $n{:}b$ from $(\langle n_1, b_1\rangle, \ldots, \langle n_J, b_J\rangle)$ - If n denotes a type W connector without a branch named b leaving it, $n_1 \ldots n_J$ denote segments or basic structures, and $b_1 \ldots b_J$ are distinct branch names, a type R connector is formed with branches, named $b_1 \ldots b_J$, leaving it and terminating on the nodes denoted by $n_1 \ldots n_J$, respectively. Each of the nodes denoted by $n_1 \ldots n_J$ is made type R.

(Table 4.2, continued)

A branch named b is also formed from the connector denoted by n to the newly formed node.

5) __Share__ n __with__ S - If n denotes an owned gps, the set of principals S becomes the share set for the gps.

6) __Borrow__ m __from__ j __as__ n:b - If p_i appears in the share set of the gps in structure g_j denoted by m, and n denotes a type W connector in g_i with no branch named b leaving it, a branch named b is constructed from the connector denoted by n in g_i to the type E connector denoted by m in g_j.

newly created or accessed procedure structures to emanate from any connector which is accessible from the root of the gprs by a path of type W. This is a generalization from the elementary environment in which procedure structures are attached to the eprs root node. This has a similar effect to going from one file directory per principal to a tree-like directory structure for each principal, as is (essentially) done, for example, in going from the M.I.T. CTSS system to MULTICS. Principals can group procedures according to common aspects, and give mnemonic significance to the path names used to name the procedures.

4.5. Properties of the General Environment

As in Section 4.2., we must show that use of the general environment mechanisms always leaves well formed general environments (abbreviated wfge) well defined, and that the requirements of Section 2.5. are satisfied.

Preserving Well Formedness

Assume that a wfge exists, and consider a particular gprs g_i. The following properties hold:

Proposition 4g.1. Use of commands by principals cannot cause any branches to be formed which terminate on the root node of g_i.

Proof: The only commands which can be used to form branches to type W connectors are *initiate process* and *form gps*. *Initiate process* can cause branches to be formed only to new type W connectors. *Form gps* can form branches to type W connectors, but only those specified by a path name. Since the root node of g_i has no branches to it, it cannot be denoted by a path name.

Proposition 4g.2. Use of commands by principals always causes g_i to be transformed into another general data structure, such that each of its type W nodes has only one branch (in G) terminating on it.

Proof: Note that the addition or deletion of branches, terminating on type R or type E nodes in g_i, which emanate from nodes not in g_i does not affect g_i, since those branches are not part of g_i.

Also the commands do not allow the alteration of type R segments or of branches leaving a type E or type R connector. Thus, we need only be concerned with the alteration of type W segments and the addition or deletion of branches leaving type W connectors in g_i.

The type W nodes of g_i can be separated into two categories: those directly accessible to p_i and those unowned nodes in gps's which are substructures of g_i. To each node in the first category there must exist a path of type W in g_i. Since each type W node can have only one branch terminating on it, there is no other path to the node, and thus it is not in the second category. Similarly, any type W node in a procedure structure must be reached by a path of type W from a type E connector, and thus all paths to the node must pass through the type E connector. Thus the node is not directly accessible to any principal. Therefore the two categories of nodes are disjoint.

Nodes of the second category are directly accessible to no principal. Thus, they can be accessed only by processes executing the procedure structures with which they are associated. As we have seen in Chapter 3, each process operating individually will transform a gps into another gps. We assume that each shared gps, whose program may be executed by more than one process simultaneously, forces the processes to interact so that they alter data associated with the gps only in ways which keep the gps properly formed. Thus use of initiate process (and terminate process) by any principal will always leave the gps used by that process well defined, and a gprs which contains that structure also will remain well defined. Also, since a process can only form branches to newly created type W nodes, the process will cause no type W node of that gps to have more than one branch terminating on it.

Nodes of the first category, those which are directly accessible to p_i, can be utilized only through commands issued by p_i, since they are in no gprs other than g_i. Since a process can only transform a gds into another gds, use of any of these type W nodes by p_i through initiate process must leave g_i well defined. Also, since a process can only form branches to newly created type W nodes, it cannot cause any of these type W nodes to have more than one branch terminating on it.

No other commands can form branches to type W segments. Form gps can form a branch to a gds root node, but it deletes the branch previously terminating on it. Branches from type W connectors can be formed through use of form gps, form bs, and borrow. However, in each case the branch terminates on the root node of a gps or bs, which are allowable substructures of g_i.

Proposition 4g.3. Use of commands by principals causes a well formed general environment to be transformed into another wfge.

Proof: The set P cannot be altered by any command. Also form gps causes the function O to be extended when a type E connector is created. These two facts, along with Propositions 4g.1. and 4g.2. imply that the use of commands always causes a well formed (well defined) general environment to be transformed into another general environment.

To see that a transformed general environment is well formed, consider a type R node in G, and additional paths which may be constructed to it. The only commands which can cause a branch to a type R node to be formed are <u>form</u> <u>gps</u> and <u>form</u> <u>bs</u>. In each case, the node must be directly accessible to the principal, say p_i, executing the command. This implies that the node must be owned by p_i (compare Definitions 4.7. and 4.9.), and since the existing environment is well formed, the node cannot be directly accessible to any other principal. Thus, <u>form</u> <u>gps</u> and <u>form</u> <u>bs</u> can only be used by p_i to construct branches to the node, and these branches must leave nodes owned by p_i. All new owned paths to the node therefore must be in g_i, and the general environment is well formed.

<u>Controlled Information Sharing Requirements</u>

This section includes theorems corresponding to the requirements of Section 2.5., which restate those requirements in terms of the structures and relationships associated with general environments.

<u>Theorem 1g.</u> Only general procedure structures can be directly accessible to more than one principal in a wfge.

<u>Proof:</u> For basic structures, general data structures, and segments, ownership and direct accessibility require the same conditions. Thus, since each of these must have a unique owner, only the owner may have direct access to them.

Theorem 2g. A principal can request the activation of any directly accessible gps, and may supply, for use during that activation of the eps program, any directly accessible gds. Information can be created during the gps execution, and any of that information, which is not part of the gps or is not destroyed during execution, becomes directly accessible to the principal requesting the execution.

Proof: The first part follows from the description of initiate process in Table 4.2. Information can be created by the process through create node, which attaches the new node to a type W connector accessible to the process. This node must be part of the gps or gds supplied to the process, from Proposition 3g.5. Thus, unless the branch is deleted during execution, the created node must be part of the gps, or part of the supplied gds which is directly accessible to the principal requesting the execution.

Theorem 3g. A principal can form a general procedure structure using any segments, general data structures, general procedure structures, and basic structures directly accessible to him. Any type W nodes associated with the gps become indirectly accessible to the principal.

Proof: The first part is true from the description of form gps in Table 4.2. Since form gps deletes the existing branch to the root node of any gds's incorporated into the gps, it can be accessed only through execution of the gps and is thus indirectly accessible to all principals.

Theorem 4g. Use of form gps, initiate process, or form bs by a principal cannot affect the directly accessible information of any other principal, except through the alteration of type W nodes indirectly accessible to both principals.

Proof: (This was shown in the proof of Proposition 4g.2.) Use of form gps and form bs by principal p_i can have no effect on $g_j (i \neq j)$ since the substructures of the newly formed structures are not affected (except for the branches to directly accessible gds's which are only in g_i. Use of initiate process by p_i must supply a gds directly accessible to p_i (and thus not directly accessible to p_j). Thus only type W nodes indirectly accessible to p_j can be affected.

Theorem 5g. Let P be a gps. During execution of the program of P, a process can access all segments directly associated with that program, all root nodes of directly associated general data structures, and basic structures, and all type R or W nodes reached by paths of type W-R in the gds supplied for that execution. Furthermore, the process may create other gds nodes and utilize them.

Proof: This is proved in Section 3.7.

Theorem 6g. Let P be a gps and Q be the gds supplied for the execution of P. During the execution of the program of P, a process can request the execution of any gps directly associated with that program or any gps, contained in Q, whose root node can be reached by a path of type W. However, while executing the program of P, the process cannot access the program of such a gps, or any of the segments or structures contained in such a gps, unless they can also be reached by paths of type W-R in P or Q.

Proof: This is proved in Section 3.7.

Theorem 7g. Each gps created by a principal becomes owned by him.

Proof: This follows directly from the description of form gps in Table 4.2.

Theorem 8g. A principal can make the use of any owned gps available to any principal or group of principals.

Proof: This follows directly from the description of share in Table 4.2.

Theorem 9g. A principal can establish direct access to any gps made available to him.

Proof: Borrow causes a branch to be formed to the gps root node from a type W connector directly accessible to the principal. Since this path is then an owned path, the gps is directly accessible to the principal.

4.6. Sharing Decisions by Algorithm in a General Environment

The sharing algorithm can be implemented in the same way as in the elementary environment case. Again, the algorithm should be able to determine the identity of the process owner through an instruction k = process owner.

The connect instruction can be utilized to allow sharing of procedures using a decision algorithm. In this case, however, it should be possible to connect the borrowed procedure to any owned, type W connector in the borrower's gprs rather than just to its root node. Thus, we assume that the node to which the procedure is to be connected is part of the data structure passed to the decision algorithm. The form of connect is now: (Assume the process is using activation map M_k.)

connect (x_i, b, x_j) - If x_i is an internal pointer to a type E connector, and if $0(f_k(x_1)) = 0(f_k(x_i))$, if $f_k(x_j)$ is an external pointer for a type W connector, and there is no branch named b leaving the connector pointed to by $f_k(x_j)$, a branch named b is formed from the node pointed to by $f_k(x_j)$ to that pointed to by $f_k(x_i)$.

Note that the node to which the procedure is connected must be pointed to by an external pointer. This ensures that the node is part of the gds originally supplied to the process, and thus is owned by the process owner. This prevents a shared gps x from "taking advantage" of a borrower's call to it by attempting to gain access for itself to another shared procedure y. This could be done by having x pass a data structure internal to it, instead of the process owner's gds, to the decision algorithm, and thus have the branch to y be connected internally to x rather than to a node owned by the process owner.

Since connect can only be used to attach branches to a data structure owned by the process owner, say p_i, the newly created paths are all in g_i, and the gps is made directly accessible only to p_i.

The only properties which are affected, of those discussed at the end of the last section, are those dealing with the mechanisms for sharing. As we indicated in 4.2., not every gps may be shared using connect, since a borrower must access the "sharing" procedure through use of borrow before connect can be used. However, this again is no restriction since any gps can be made a substructure of another gps, which may in turn may share the first gps through use of connect.

CHAPTER 5

CONTROLLING THE USE OF SHARED INFORMATION

5.0. Introduction

In the last two chapters, two models have been developed which represent methods for organizing and using stored information. These models allow each principal to share with others the use of procedures which he created, and therefore owns. This implies that he is able to share the (indirect) use of information he has borrowed from others.

It is the purpose of this chapter to investigate the implications of this sharing strategy, and to describe other types of facilities which would allow the owner of a procedure greater control over its use. The general approach of the discussion is to present, for each type of sharing strategy, an example in which certain types of control over the use of shared information are required. Then a description of modifications to the model, which would implement this strategy, are discussed, and arguments demonstrating their validity are given.

Section 5.1. discusses the facilities supplied by the well formed general environment of Chapter 4. In the remainder of the chapter, other types of sharing facilities are discussed which allow finer degrees of control, but which can be implemented only through additional, more complex mechanisms.

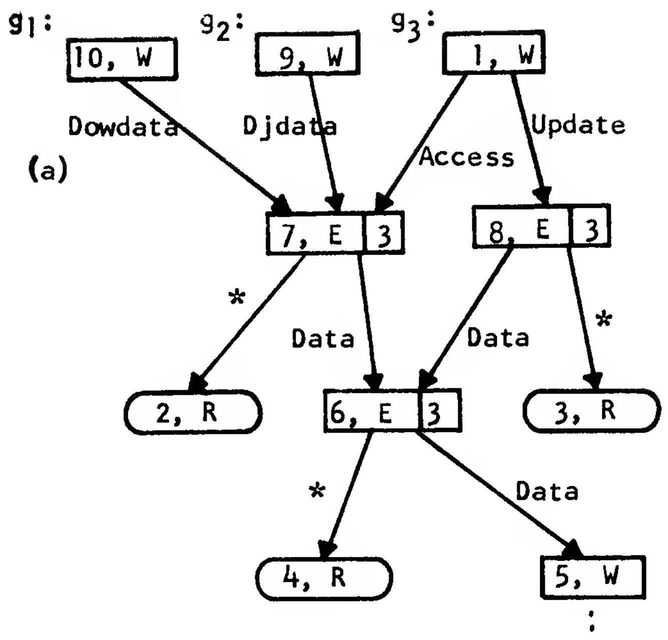
5.1. Information Sharing in a General Environment

An Example

Let us assume that there exists a principal named Dowjones. Dowjones has constructed a data base and programs for maintaining the data base. The information in the data base consists of certain information regarding stocks, say current price and the size and price of recent transactions. Dowjones wishes to provide access to the information of the data base to his customers and to simultaneously update the data base so it remains current.

Two of Dowjones' customers are Trendfinder and Chartist. Both of these customers provide services to investors, consisting of various types of charting and statistical information on particular stocks. The manner of implementation of the services by Trendfinder and Chartist involves use of proprietary programs which retain knowledge regarding past trends of stocks and utilize the Dowjones data base for current information.

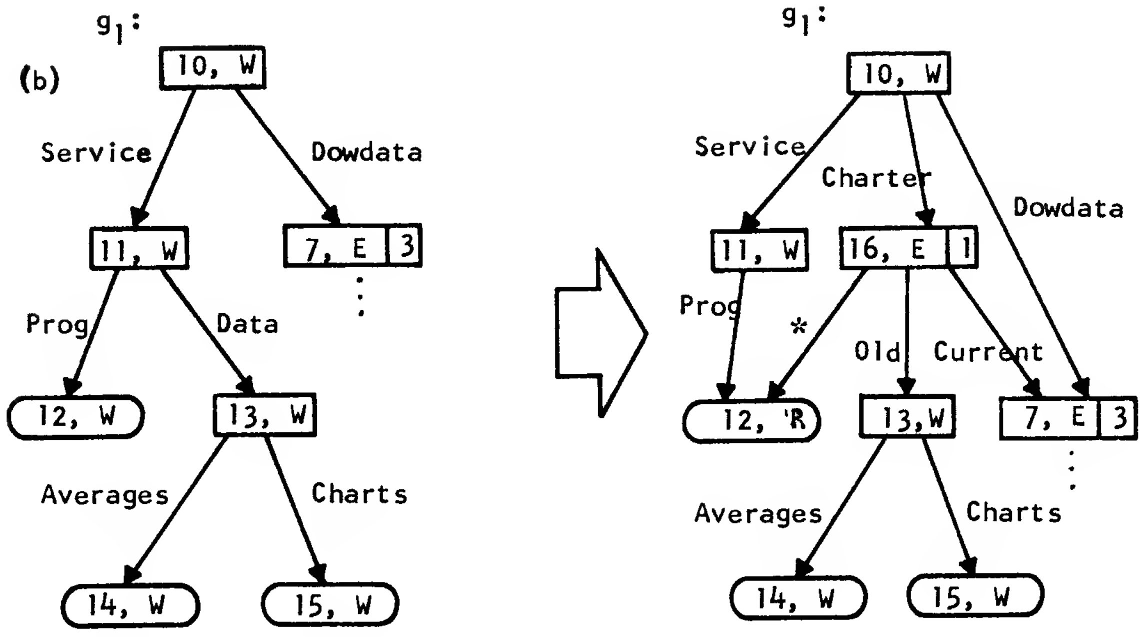
A possible structuring of the information required for these services is shown in Figure 5.1. Part (a) of the figure shows the structure of Dowjones' service. A caretaker procedure (node 6 is its root) coordinates accesses and alterations to the data base. The procedure, which is made available to customers, is named "Access", and allows the customers' processes to access the data base. Maintenance of the data base is performed through another procedure, "Update", which can only be used by Dowjones. Figure 5.1.(a) also shows how the customers Trendfinder (p_2) and Chartist (p_1) have obtained access to



p_3: <u>Share</u> Access <u>with</u> $\{p_2, p_1\}$

p_1: <u>Borrow</u> Access <u>from</u> p_3 <u>as</u> Dowdata

p_2: <u>Borrow</u> Access <u>from</u> p_3 <u>as</u> Djdata



p_1: <u>Form gps</u> Charter <u>from</u> (Service.Prog, ⟨Service.Data, Old⟩ , ⟨Dowdata, Current⟩)

Figure 5.1. An Example of Sharing in a General Environment

the data base from Dowjones (p_3) through use of share and borrow.

Part (b) of Figure 5.1. shows how Chartist makes use of the Dowjones service in supplying his own. His service, provided by the procedure named "Charter", maintains old information in the gds it calls "Old" (with root node 13) and makes use of the Dowjones service through a branch named "Current". Principal Chartist is then free to make its "Charter" service available for use by others.

We have assumed that Dowjones does not care to whom Chartist and Trendfinder make available their services. If we assume that Dowjones charges its customers for its service on the basis of amount of usage, this is not an unreasonable assumption. The more customers Chartist has, the more the Dowjones service is used. In the wfge, when a customer of Chartist, named Investor, requests service, it is Investor's process which actually "uses" the Dowjones service, not a process of Chartist. It is not reasonable, however, to require that Dowjones know of each of Chartist's customers and charge them for this usage. Rather, it is Chartist who should be held responsible for the use made by his service of the Dowjones service. Chartist should therefore pay for this usage, from the charges he received for his services from his customers, such as Investor.

Therefore, we require that it be possible for the system to determine to whom charges for usage should be made. As Investor's process is executing the procedure of the Chartist service and makes use of the Dowjones service, responsibility for that usage should be given to Chartist, since it is he to whom Dowjones rents his service. This information can be extracted easily from a record of procedure calls. Since Chartist can only share owned procedures, he cannot allow a customer of his to use the Dowjones service directly. Thus all calls to Dowjones' service, which are made by processes owned by customers of Chartist, must be from procedures which are owned by Chartist. Thus, the requirement that only owned procedures may be shared ensures that only a procedure owned by a customer of Dowjones may call the Dowjones procedure, and therefore it may be determined which of Dowjones' customers is _responsible_ for the usage.

To this extent, then, the wfge provides control of information sharing. However, there are cases in which more control is required. The next section provides a discussion of such a case.

5.2. Restricted/Unrestricted Sharing

An Example

Let us consider, again, a case in which a data base is maintained for use by others. However, assume that the data base contains personal medical records. This centralization of records could enable a patient to have his complete medical history made available to any doctor quickly and easily, and would relieve doctors of the problems of keeping complete and up-to-date records. In this case, it is clear that the "propagation" of use must be controlled.

To illustrate this point, let us assume that a principal, Medbank, has developed a set of procedures for maintaining this data base, which are shown in Figure 5.2. We ignore problems dealing with identification and authorization, except to say that there exist three types of principals which desire to access the records: doctors, patients, and researchers. Doctors should be able to access and alter the records of patients (who have authorized the doctor in some way, perhaps within their record). Each patient should have free access to, but limited alteration capability for, _just_ his record. Finally, researchers could have free access to certain (non-identifying) portions of patient records.

Let us assume that, in order to ensure correct control of access, the doctors and patients must be _unable_ to grant to others indirect use of their abilities to use the data base. This could be accomplished by restricting users of the "Doctors" and "Patients" procedures from sharing with others any procedures that use those as subprocedures, and by not restricting the users of the "Research" procedure.

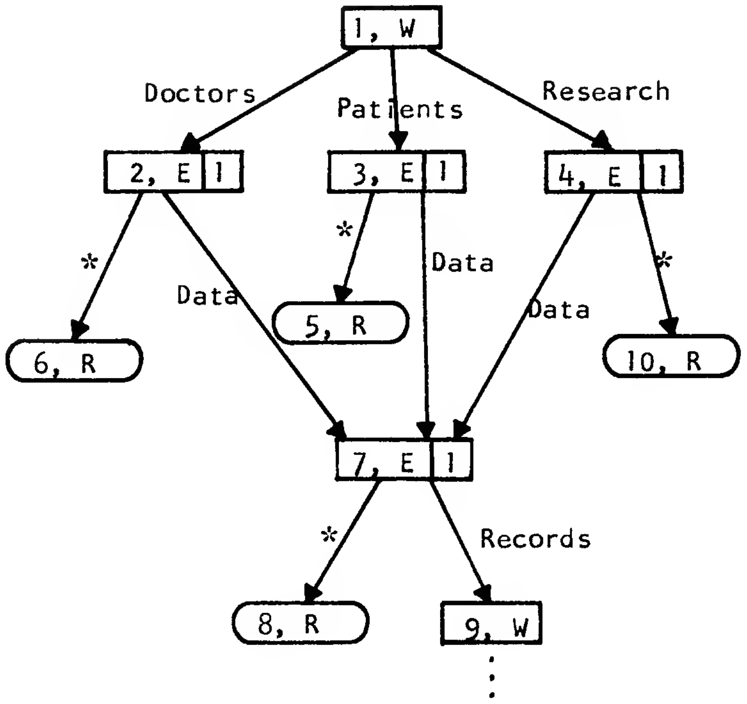


Figure 5.2. An Example of Restricted/Unrestricted Sharing

<u>An Implementation</u>

This type of control of sharing could be provided by performing the following modifications to the general environment and its associated commands.

1) Associate with each branch of each g_i a class, which can be U (for unrestricted) or R (for restricted).

2) Augment the definition (in Table 4.2.) of <u>form</u> <u>gps</u> so that

   a) the newly formed *-branch becomes class U,

   b) for all i, the newly formed branch of name b_i takes on the class of the last branch in path n_i, and

c) the class of the branch terminating on the newly formed node becomes U if and only if all branches leaving the node are of class U; otherwise the branch becomes class R. If the branch becomes class R, the class of the branches in the path named n also becomes R.

3) Augment the definition of <u>form</u> <u>bs</u> so that all of the newly formed branches are of class U.

4) Change the definition of <u>share</u> to be

<u>share</u> n <u>with</u> S - If the last branch of n is of class U and denotes an owned gps, S becomes the share set for the gps. The elements of S are ordered pairs, each pair consisting of a principal and either a U or R restriction.

5) Augment the definition of <u>borrow</u> so that the newly formed branch takes on the class associated with p_i in the share set of the borrowed procedure. If that class is R, all of the branches on the path n also become class R.

6) Augment the definition of the instruction <u>create</u> <u>node</u> so that it always creates branches of type U.

To show that these modifications accomplish the desired objectives, the following two properties are demonstrated.

<u>Proposition 5.1.</u> If a principal is allowed restricted access to a gps, he cannot share any owned general procedure structure which contains that gps.

Proof: First we inductively show that all paths in the borrower's structure to the root node of the borrowed gps are composed solely of class R branches. Let p_i be the borrower of a gps A.

Basis: The first path formed in g_i to the root node of A is composed solely of class R branches. This path must be formed through use of borrow, since it is the only command which does not use only paths within g_i. From the form of borrow, we see that all of the branches in the path composed of path n and branch b are made to be class R. Also, since path n must be the only path to the type W connector on which it terminates, that path (path n and branch b) must be the only one formed to the root node of A by borrow, and thus must be the first.

Inductive Step: If the first k paths to the root node of A are all composed of class R branches, so is the k+1 path. This new path could be formed only using borrow or form gps. (The only other command which forms new paths to existing nodes is form bs, but those paths can only be to type R connectors or segments.) If borrow is used, the above argument applies. If form gps is used, it may form a path to A by incorporating A or a gds containing A into a new gps, say B. In either case, the path used to specify this structure must be at least part of a path to A and must be composed of class R branches. Thus the branch formed from B's root node to the structure containing A must be of class R, as must the branch to the root node of B and the path to the type W connector from which that branch emanates. Thus, the new branches of the new path to A must all be of class R, and the entire path must be composed solely of class R branches.

Thus we have shown that all paths to the root node of A must be only class R branches. From the definition of share, we see that a branch to root node of a gps must be of class U to be "shareable". This is not possible for any node on a path to A, and thus no gps containing A may be shared.

Proposition 5.2. If use of none of the components of a gps has been restricted by another principal, the gps may be shared by its owner.

Proof: First note that branches formed by create node and form bs are of class U, and that form gps and borrow only change the class of branches to type W connectors. Thus, all branches terminating on segments or type R connectors must be of class U.

As a gps is formed, the classes of branches from its root node to its components reflect the classes of other branches to the components. From the above argument, it is clear that the branches to basic structures and segments must be of class U. Similarly, the branches to borrowed gps's are of class U if there is no restriction, from the form of borrow.

Other components can be general data structures and other owned gps's. The branch to the gds was created as class U and could only have been changed if it contained a gps whose use was restricted. If we assume that its use has not been restricted, then the branch to the gds must therefore be class U.

By repetition of this argument, we can see that the branch to any owned gps must be of class U if it has no components whose use has been restricted. Thus, all of the branches leaving the gps root node are of class U, and the branch entering is as well. Thus, share can be utilized for this gps.

Thus the mechanisms work as desired. Returning to the Medbank example, we see that if the share sets for the "Doctors" and "Patients" procedures contained principal names with R indications, those principals could only use the procedures themselves. However, if the principals in the share set for "Research" were given U indications, they could share the results of their research with others by sharing procedures using the "Research" procedure.

This type of Restricted/Unrestricted sharing, however, does not allow the restrictions on a procedure's use by a particular principal to be made except in an "all or nothing" manner. The next section discusses mechanisms for allowing selective removal of restrictions.

5.3. More Selective Sharing Restrictions

An Example

Let us suppose that a principal, named Simulex, has developed a simulation language called Simlang and a processing program for running simulations written in that language. His primary goal is to use this language in providing modelling services to certain groups of principals with particular needs, e.g. urban planners, civil engineers. Simulex is also willing to make the use of Simlang available to others. However he does not want his customers, who have access to the language, to use it to provide services which might compete with his own services.

The mechanisms of the last section could provide a solution to this problem, by enabling Simulex to allow others only Restricted access to the simulation language. However, let us assume there exists another principal, called Modelex, who wishes to use Simlang in offering a modelling service which would not be in competition with any of Simulex's services. Modelex is willing to satisfy various conditions imposed by Simulex concerning, for example, marketing and advertising policy, potential customers, and charging policy, in return for the ability to market his service (which uses Simlang).

In order to accomplish this sharing, it must be possible for Simulex to allow Modelex to make available this particular application of his procedure, Simlang, without allowing Modelex to make available other uses of Simlang. An example of this situation is shown in Figure 5.3. Simulex would like to allow Modelex to share the gps called Transport (root node 4) without enabling him to share Simlang (root node 2) or any other procedures containing Simlang, such as Civilengr (root node 8).

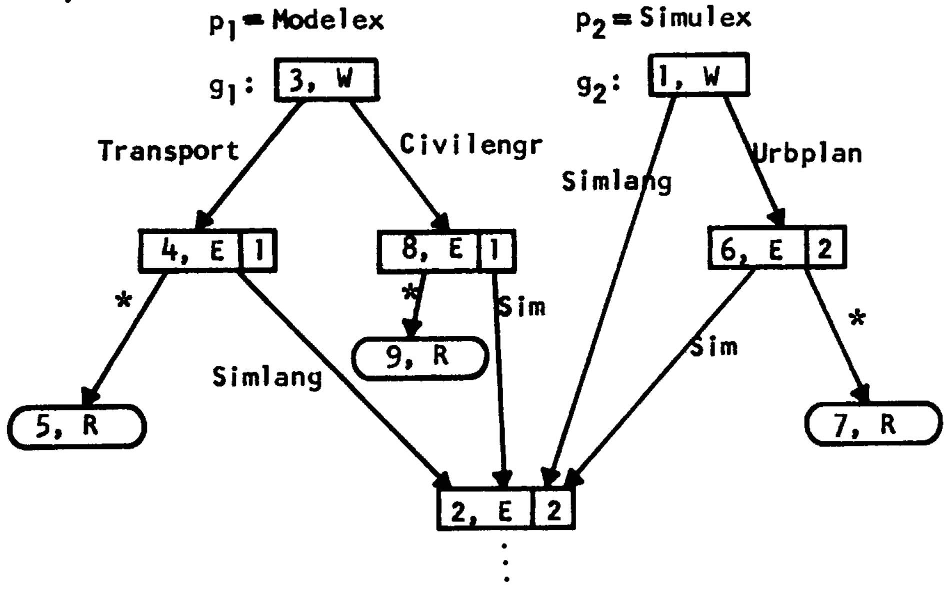


Figure 5.3. An Example of Selectively Restrictive Sharing

The General Problem

This example represents a general class of sharing controls. In each case, it must be possible for the owner of a shared procedure to allow a borrower to "propagate" use of certain procedures utilizing the shared procedure without allowing him to share others which also use the shared procedure. The members of the class can be differentiated by the conditions which may be enforced by the owner of the shared procedure on the propagation of its use. These conditions might be enforced within the utility, for example, by restricting potential customers or charging policies; they might exist, and be enforced, outside the utility, such as by restricting advertising policy regarding certain characteristics of the service provided.

Nature of the Solution

Although the utility may not provide enforcement of all of the conditions of a contract, it should be able to allow use of the "selective" restrictions characteristic of these sharing controls. These can be implemented using a generalization of the mechanisms discussed in Section 5.2.

The strategy is to associate a set of conditions with each gps root node, for example the names of all principals who are restricting the use of that structure. When a new gps is created, its "condition set" is formed by taking the union of all of the condition sets of its components. In order to share a gps, all of the conditions in its "condition set" must be satisfied.

In order to allow the selective propagation of use, a capability could be provided to allow the removal of a condition by the principal which imposed it. Each element of the condition set can have a principal associated with it, and only that principal may remove the condition. Then, by allowing a principal to remove the condition in only selected sets, the sharing of particular gps's would be possible.

For example, in the situation described in Figure 5.3., Simulex would place a restrictive condition on the use of Simlang. This condition would then be "propagated" into the condition sets of Transport (node 4) and Civilengr (node 8), owned by Modelex. Since he established that condition, he could then remove the condition for Modelex's Transport, while leaving it on the others. Modelex could then share Transport, and that condition would not affect gps's containing Transport, but the condition would continue to restrict other uses of Simlang.

In summary, the following steps are required for this general solution.

1) Associate a condition set with each gps root node.

2) Augment the definition of <u>form gps</u> so that it creates a condition set for the new node containing all conditions affecting the gps's which are directly associated with it (or are contained in directly associated gds's).

3) Augment <u>share</u> so that it allows sharing only if the conditions associated with the gps are satisfied, and so the gps owner may insert additional conditions.

4) Provide a means for the principal which imposed the condition to remove it for any individual case.

5.4. Sharing Decisions by Algorithm

The connect instruction proposed in Chapter 4, to allow the procedure owner to specify an algorithm by which decisions are made to share the procedure, could be adapted to implement either of the new sharing strategies described above. In each case, connect would have to be modified to have the same effect as borrow.

C H A P T E R 6

RELEVANT MISCELLANY

6.0. Introduction

The general environment model involves a number of assumptions and simplifications regarding the ways in which information can be stored and used within a computer utility. Some of these are concerned with the type of control which an owner should have over shared information. Chapter 5 explores some of the consequences of these assumptions and describes some modifications to the general environment model which allow more extensive control of shared information.

In order to concentrate on the issues of information sharing, a number of other simplifications were made. It is assumed that there can exist at any time no more than one process for each principal. The procedures which may be activated by a process must be directly associated with the procedure, or be part of its data structure, so that recursive procedure calls and certain types of argument passing are not possible. Also, some facilities which would be necessary in any utility implementation are not discussed, such as editing facilities for structures and mechanisms for revoking access abilities.

In this chapter, mechanisms for removing some of these simplifications are discussed. It is the intention of this chapter to demonstrate that additional capabilities can be added to the basic model in a way that is consistent with the mechanisms of the model, and that does not destroy its basic properties.

6.1. Parallel Processing

In this section we consider a generalization of the general environment of Chapter 4 to enable a single principal to perform processing activities in parallel by allowing him to own multiple processes which exist concurrently. The two problems considered are 1) how multiple processes can be initiated, operate, and be terminated without conflict while using portions of the same general principal structure (g_i) concurrently, and 2) how communication among these processes may take place.

Multiple Processes

The method used to allow a principal to own multiple processes is to enable a process to initiate another. A principal initiates a computation* by initiating a process, as in Chapter 4. The process may initiate further processes which may operate concurrently with it in a non-conflicting manner (see next paragraph). The computation is terminated only when all of its processes have been terminated.

We consider two processes to be operating in <u>conflict</u> if they simultaneously have the ability to access a type W node owned by a principal. Thus operation without conflict requires that simultaneously existing processes use disjoint, that is disjoint relative to owned type W nodes, portions of a gprs. Note that this does allow processes

---

* We are here giving the term "computation" a meaning, essentially that of Dennis and Van Horn [11], of "a set of processes that are all working together harmoniously on the same problem or job".

to utilize a commonly accessible general procedure structure, which may contain alterable, but unowned, data.

The process which initiates another process passes to the new process a portion of its gds. To avoid conflicts, it must be ensured that the initiating process retains no pointers to the type W nodes used by the initiated process, and that it is unable to obtain such pointers. To accomplish this an identifier is associated with each process, and a process identifier ID and a pointer count C are associated with each owned, type W node. As we shall see, ID is used to indicate which of the owner's processes has access to the node, and C is used to indicate the number of pointers in the state of the process which point to the node.

Table 6.1. contains a description of an instruction set which enables processes to initiate others, and to operate concurrently in a non-conflicting manner. The first five are analogues of the instructions used in the general environment, but the last two instructions, _initiate process_ and _remove pointer_, are new. _Initiate process_ enables one process to initiate another, which is established to execute a gps contained in one of the structures available to the initiating process, while using as its gds a portion of the gds supplied to the initiating process. _Remove pointer_ enables a process to remove a pointer from an activation; this allows a process to relinquish its ability to access a type W node so that another process may then access it.

External/internal indications are again assumed to be associated with pointers to type W connectors. Since the external pointers

TABLE 6.1.

Assume that process h is using map M_k as it executes the instruction.

1a) x_j= <u>obtain</u> <u>node</u> (x_i, b) - If $f_k(x_i)$ is a type CR pointer or an internal type CW pointer, and a branch named b leaves the connector pointed to and terminates on a node y, then $f_k(x_j)$ becomes a pointer, internal if $f_k(x_i)$ was, to y.

1b) x_j= <u>obtain</u> <u>node</u> (x_i, b) - If $f_k(x_i)$ is an external type CW pointer, and a branch leaves the connector pointed to and terminates on a node y, and $C(y) = 0$ or $[C(y)>0$ and $ID(y) = h]$, then $f_k(x_j)$ becomes a pointer to y, $C(y)$ is incremented by 1, and $ID(y)$ h. If x_j had a previous value of an external pointer to a type W node, then the pointer count for that node is decreased by 1.

2) x_j= <u>create</u> <u>node</u> $(x_i, b, \left\{\begin{matrix}\text{segment}\\ \text{connector}\end{matrix}\right\})$ - If $f_k(x_i)$ is a type CW pointer and no branch named b already leaves the connector pointed to, a branch with name b is created from the connector to a newly created type W node, y, of class specified by the third argument, and $f_k(x_j)$ becomes a pointer to the new node. If $f_k(x_i)$ is external, so is $f_k(x_j)$, and $C(y) = 1$ and $ID(y) = h$. If $f_k(x_i)$ is internal, so is $f_k(x_j)$. If x_j had a previous value of an external pointer to a type W node, then the pointer count for that node is decreased by 1.

3) <u>call</u> (x_i, x_j) - If $f_k(x_i)$ is a type CE pointer to a connector z, and $f_k(x_j)$ is an external type CW pointer to a connector y,

(Table 6.1. continued)

then $C(y)$ is incremented by 1, and a new activation map M_{k+1} is established such that $f_{k+1}(x_1)$ is a pointer to z, $f_{k+1}(x_0)$ is a pointer to the program segment associated with z, and $f_{k+1}(x_2)=f_k(x_j)$.

4) <u>return</u> - If $k>1$, M_k is deleted and M_{k-1} is reactivated. If $k=1$, the process is terminated. In either case, the pointer counts are adjusted downward for the nodes pointed to by external type CW pointers in M_k.

5) <u>delete</u> (x_i, b) - If $f_k(x_i)$ is a type CW pointer to a connector with a branch named b leaving it, that branch is deleted from the structure.

6) <u>initiate process</u> $(x_i, x_j{:}b)$ - If $f_k(x_i)$ is a type CE pointer to a connector w, and $f_k(x_j)$ is an external type CW pointer to a connector y, and if the branch leaving y named b terminates on a type W connector z with $C(z)=0$, then a process is established with identifier g. The activation map M_1 of g is formed so that $f_1(x_1)$ is a pointer to w, $f_1(x_0)$ is a pointer to the program segment associated with w, and $f_1(x_2)$ is a pointer to z. Also $ID(z)=g$ and $C(z)=1$.

7) <u>remove pointer</u> (x_i) - If $f_k(x_i)$ is an external pointer for a type W connector y, then $f_k(x_i)$ is deleted from M_k and $C(y)$ is decremented by 1.

indicate which of these connectors are owned by the process owner (the internal pointers denoting type W connectors associated with gps's), these pointers also indicate the type W nodes having associated ID and C indications.

Note that in two instructions, _obtain node_ and _initiate process_, conditions regarding the ID and C indicators of a node to be accessed must be satisfied before the instruction can be executed. These conditions do not relate to keeping the structures well defined, but are required to ensure that only one process may have the ability to access the node under consideration. We assume that if these conditions are not satisfied when execution of the instruction is attempted, it will be at some future time when another process "is finished" with the node.

Thus we require that these instructions be implemented so that the process waits for the conditions to be satisfied. That is, if a process attempts to execute an _obtain node_ or _initiate process_ instruction, and all of the conditions for the execution are satisfied except for the required values of ID and C for a type W connector to be utilized, the activity of the process is suspended. When the conditions become satisfied, the process activity is resumed, and the instruction is executed. This type of implementation is necessary since we have provided no other means for synchronizing the activities of concurrently existing processes. In the next section, we shall see how this implementation allows _obtain node_ to be used to easily provide interprocess communication.

The commands _initiate process_ and _terminate process_ in the general environment must be replaced by _initiate computation_ and _terminate computation_. They are described as follows, assuming they are issued by principal p_i:

initiate computation $(m, \left\{\begin{matrix} n \\ - \end{matrix}\right\})$ - If m denotes a gps and n a gds, and if there exist no processes owned by p_i, a process owned by p_i is initiated using the designated structures (a blank second argument denoting g_i). If the identifier of the new process is h and the root node of the structure denoted by n is y, then $ID(y)=h$ and $C(y)=1$.

terminate computation - All processes owned by p_i are terminated.

These commands start a computation by initiating a single process, leaving it to the process to initiate others, and, if termination is required, terminate all processes of the computation at once. The other commands of Figure 4.2. need not be changed.

Properties of Concurrently Active Processes

To show that these mechanisms operate correctly, we must argue that

1) no two processes may have pointers to the same owned, type W node simultaneously, and
2) for any owned type W node, only one process may attempt to gain access to it at any time.

The first point tells us that no conflicts can arise if the instructions operate correctly. The second tells us that only one process has the ability to request access to any type W node at any given time, and thus no races between requests can arise.

The first point follows directly from the description of the instructions. Each instruction, causing a pointer to an owned type W node to be created, increments the appropriate C by 1, each one causing pointers to be deleted decreases each C, and create node initiates C of a new node at one. Thus the pointer counter C indeed represents the number of existing pointers to that node. Also, the use of ID by obtain node and call implies that, if there already exist pointers to an owned, type W node, already having the pointers. Thus if any process has pointers for a node, it is the only one.

The second point follows from the first and the fact that there exists one path, of type W, to an owned, type W node. Thus, there exists exactly one owned, type W connector from which the node can be reached, by use of obtain node or initiate process. Only one process may have access to that node, and thus only that process can request access to the reached node.

Communication Between Processes

We could constrain communication between processes to take place through commonly accessible procedures as we do in the case of processes owned by different principals. However, in this case, processes can be extremely closely related, being parts of the same computation, and a simpler, more direct method of communication could be justified among such processes.

Fortunately, the mechanisms just described provide the capabilities needed for communication among processes of a computation. An example should illustrate how these mechanisms can be used in this manner.

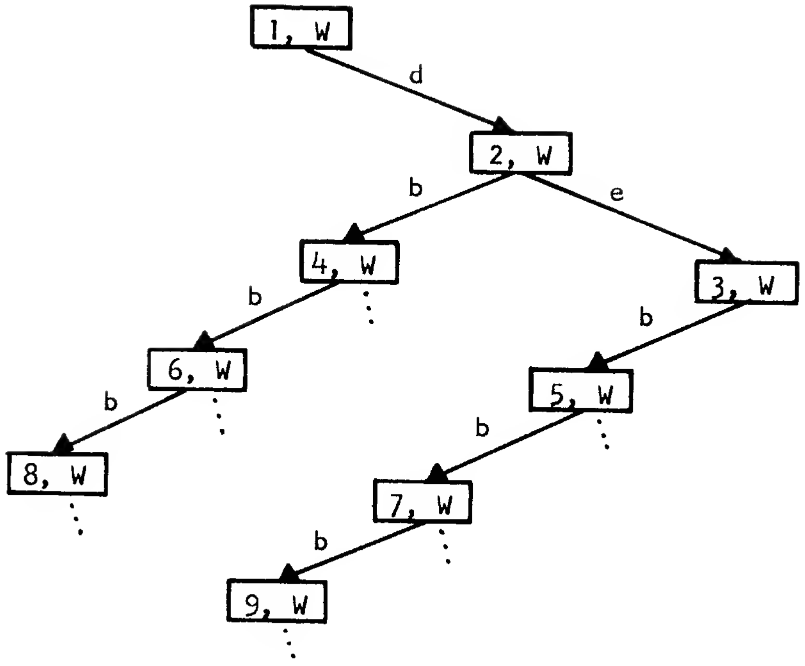
Consider a process A which has constructed the general data structure shown in Figure 6.1.(a). We assume that only the nodes numbered 1-5, and the branches between them, currently exist. Process A removes all of its pointers to nodes 2, 3, and 4, (thus reducing C(2), C(3), and C(4) to zero), while retaining pointers to 1 and 5, and initiates a second process, say B, giving it the use of node **2** as the root node of its gds.

To complete the initialization for communication between the two processes, process B obtains pointers to nodes 3 and 4 and removes its pointers to node 2. Process A then obtains a pointer to node 2. (Note that A could attempt to obtain this pointer at any time, and its activity would be suspended until process B had removed all of its pointers for node 2 and reduced C(2) to zero.)

After this initialization, the situation is as shown in the top of Figure 6.1.(b). For simplicity, we have assumed that both processes are utilizing activation map M_1 when the communication occurs. In those maps, process A has variable names u and v associated with type CW pointers to nodes 2 and 5, and B has r and s similarly associated with nodes 3 and 4.

Communication between the processes can be accomplished through

(a)



(b)

Process A

M_1:

u	2, CW
v	5, CW

A → B v = <u>create</u> <u>node</u> (v, b, CONNECTOR)

u	2, CW
v	7, CW

B → A u = <u>obtain</u> <u>node</u> (u, b)

u	4, CW
v	7, CW

Process B

M_1:

r	3, CW
s	4, CW

r = <u>obtain</u> <u>node</u> (r, b)

r	5, CW
s	4, CW

s = <u>create</u> <u>node</u> (v, b, CONNECTOR)

r	5, CW
s	6, CW

Figure 6.1. Communication between Two Processes

use of _create node_ and _obtain node_, as shown in Figure 6.1.(b). For example, A can make a message available to B by attaching a structure containing the message to the node associated with v (node 5 in the example), removing its pointers for that structure, and executing the instruction v=_create node_ (v, b, connector). Use of this instruction causes the pointer $f_1(v)$ to be changed to point to node 7. Also, assuming that the previous value of v was the only pointer to node 5, the pointer count C(5) is reduced to zero. Through execution of r=_obtain node_ (r, b), process B can then gain access to node 5. (Again, process B could attempt to execute this instruction at any time, and its activity will be suspended until process A has made node 5 available.) Once it has obtained access to node 5, process B can access the message attached to that node, but it cannot traverse branch b to node 7 until A has made it available.

Repeated use can be made of these instructions for passing other messages from A to B. This will result in a "chain" of nodes being formed (shown as nodes 5, 7, 9 in Figure 6.1.(a)), with a new node in the chain being used for each message. Note, however, that after B accesses node 5, he can delete the branch leaving node 3. Then, when he is finished with node 5 and has removed all pointers to it, the node is automatically deleted. In a similar way, the structure used to pass each message can be discarded when the contents of the structure are no longer needed by the process which received the message.

As is shown in the final part of Figure 6.1.(b), an analogous method can be used for passing messages from B to A. In this case, the chain of nodes leaving node 2 in Figure 6.1.(a) is used. Note that process A could accomplish communication with other processes it created through a similar mechanism, as could B with processes it created, ...

Since all of this information is contained in a general data structure which is directly accessible to the owner of the computation, it must be contained only in the owner's gprs. Thus, it is not accessible to any other principals, and the properties of the general environment are not affected.

6.2. Passing Arguments between Programs

During the execution of a program a process has available to it, in addition to the information directly associated with the program by its gps, the information in the gds supplied for this activation. Through use of commands, a principal may incorporate directly accessible general procedure structures and basic structures into a gds, and can supply this gds to a process he initiates. Thus, the _principal_ can make any directly accessible information available to the process for execution of the gps program. However, when the _process_ wishes to call a subprocedure, it does not have the ability to make all of the information, which is currently available to it, accessible during execution of the subprocedure. In particular, it cannot pass use of its directly associated information to the subprocedure.

Consider a situation in which a process is initiated, by principal p_1, to execute a procedure A. Assume that the program of A has directly associated with it two subprocedures, B and C. Since the process cannot attach gps's to a gds, it cannot call B and obtain the ability to call C during the execution of B's program, and, in fact, cannot obtain the ability to access any of the information directly associated with the program of A while executing B's program. This limitation prevents, for example, the recursive use of procedures (e.g. X calls Y, Y calls Z, Z calls X).

A modification to the general environment can be made, however, which easily removes this limitation. This modification changes the _call_ instruction to be the following: (Assume that it is executed using activation map M_k.)

call x with $(y_1, y_2, \ldots, y_j)$ - If $f_k(x)$ is a type CE pointer, and $f_k(y_1) \ldots f_k(y_j)$ are pointers, a new activation map M_{k+1} is established such that $f_{k+1}(x_1)$ is a type CR pointer for the node specified by $f_k(x)$, $f_{k+1}(x_0)$ is a type SR pointer for the program segment of the called gps, $f_{k+1}(x_2) = f_k(x)$, and $f_{k+1}(2+i) = f_k(y_i)$ for $1 \leq i \leq j$.

Also the initiate process command is changed to be

initiate process $(m, n_1, \ldots, n_j)$ - If path name m denotes a gps and path names $n_1 \ldots n_j$ denote other structures or segments, and if there exist no processes owned by p_i, a process owned by p_i is initiated such that $f_1(x_1)$ is a type CR pointer for the node denoted by m, $f_1(x_0)$ is a type SR pointer for its program segment, $f_1(x_2)$ is a type CE pointer for the node denoted by m, and $f_1(x_3) \ldots f_1(x_{j+2})$ are pointers to the structures and segments denoted by $n_1 \ldots n_j$.

These modifications allow two things. First of all, they enable the principal or process to supply pointers to any information currently accessible to the principal or process to the procedure to be executed. Also, they provide a process with a type CE pointer for the root node of the gps it is now executing (in addition to the type CR pointer). This enables the program to call itself or pass that ability on to a

subprocedure. Thus a "program" can pass to a "subprogram" any informa-tion which is "accessible" to the program.

6.3. Facilities for Utilizing General Structures

The commands and instructions of the general environment have been developed to provide the capabilities necessary to implement controlled sharing of information. It is not assumed that users of the system would be forced to directly use these mechanisms, but that higher level facilities would be developed utilizing these basic mechanisms. One important capability which these facilities should provide users is the ability to edit and debug procedures utilizing a high level language. In this section, we discuss some of the issues involved in providing these capabilities.

These facilities must operate within the restrictions of the general environment, among them being the fact that once information is utilized in unalterable form, it cannot be altered again. After a segment becomes type R, its contents cannot be changed. After a type R or E connector is formed, the branches leaving the connector may not be altered.

Under these constraints, the process of editing a procedure would involve the construction of a new procedure utilizing some new components and some of the components of the previously existing procedure. The editing procedure could ensure that paths are retained to the components of a gps so that its structure could be "copied" using the mechanisms we have described. However, in order to actually construct the new procedure, it would be necessary to perform activities which we have constrained to be performed only by commands (<u>form</u> <u>bs</u> and <u>form</u> <u>gps</u>).

These activities have been constrained to be commands to avoid certain complex problems. First of all, since _form gps_ may cause a type W segment to be changed to type R, it must be true that no type SW pointers for that can still exist. Also, the newly formed gps can be composed only of components which are directly accessible to the owner, so that the sharing mechanism (_share_ and _borrow_) is not circumvented.

However, it is clear that under the appropriate constraints, _form gps_ and _form bs_ could be utilized as instructions. An indication of whether the structures being utilized by the process are directly accessible to the process owner could be provided by keeping track of whether an owned path was followed or not. In addition, since only the process forming the new structure could have type SW pointers for the nodes changed to type W, and these pointers would be destroyed when the process is terminated, this limited ability to alter an "unalterable" segment might be acceptable.

Another issue which must be resolved concerns the privacy of information. These editing and debugging facilities will be provided by one or more principals and borrowed by principals wishing to utilize them. Since these facilities will be used for working with a principal's procedures, he must be sure that the editing procedure does not "remember" information about the procedures it is used to construct so that the owner of the editing procedure can later extract this information. It seems clear that, if a procedure is allowed to retain _any_ information regarding its activations, it may be capable of retaining an arbitrary amount of information. Thus, it would seem to be necessary that the editing and debugging procedures be constrained to retain no information regarding their activations.

One straightforward way to implement this would be to allow both general and elementary procedure structures to be constructed. When a procedure is borrowed, the borrower could be informed of the procedure type. Also eps's could only be formed using other eps's and type R segments. Since eps's can retain no information regarding their activations, editing procedures could be implemented as eps's, and the borrowers could be confident that it could not "remember" information concerning the procedures it is used to construct.

While this discussion undoubtedly has not considered all of the issues involved in providing high-level user facilities, it hopefully has demonstrated that the basic mechanisms of the general environment can be easily modified to provide required facilities without destroying the basic properties of a wfge.

6.4. Removal of Access Abilities

As we indicated in Chapter 2, it is necessary that it be possible to remove established access abilities, although it can only be done under special circumstances. Two types of removal seem to be necessary.

The first would result from the discovery that a particular procedure performed incorrectly, so all access to the procedure should be prevented. This could be done by destroying the root node of the procedure in some manner outside the mechanisms of the model, and causing an error to occur when any attempt to access that node is made.

A question remains, however, involving the fate of the contents of a general procedure structure whose root node has been destroyed. There could have existed paths to the type R and type E nodes of the gps not passing through the destroyed node, and their contents could remain accessible. However, the general data structures, directly associated with the procedure's program, could only be reached by paths through the gps root node. Thus, when that node is destroyed, these type W nodes would become inaccessible unless other provisions were made. Since this information in general might be of great value for determining what was wrong with the procedure, if not actually being an irreplaceable collection of information, it should clearly be retained. We therefore assume that, as part of the destruction of the procedure, the gds components, of a gps which is destroyed, be made directly accessible to a particular (system) principal by having them attached to his gprs. He could then make some or all of this information available to the users of the destroyed procedure as is deemed appropriate by an authority existing outside of the system.

The second type would involve the removal of a particular principal's ability to access a particular shared procedure. A more complex operation is required here, since that procedure may be used as a part of many different procedures. What is required is an ability to search the nodes of the principal's structure which are <u>directly accessible</u> to him using a method similar to that of 6.2. The search could destroy the branches connecting nodes owned by the principal to the root of the shared procedure structure. If each of these branches were destroyed, the access abilities resulting from the borrowing of the shared procedure would be removed.

Since both of these operations would require action by an authority superior to a principal, the actual mechanisms can be considered part of the design of an implementation of the model, and they will not be discussed further in this thesis.

C H A P T E R 7

IMPLEMENTATION OF A GENERAL ENVIRONMENT

7.0. Introduction

In the development of the general environment and extensions of it, we have not considered any of the issues regarding its implementation. Rather we have discussed the mechanisms in terms of their satisfaction of requirements for the controlled use of shared information. In this chapter, we briefly consider some of the issues which would have to be dealt with in an implementation of a general environment.

We assume that the utility is implemented as a large set of operating system programs which are executed by processors similar in design to general purpose processors existing today. Since the general environment is a description of a method for structuring and utilizing stored information, we consider only portions of the system which are concerned with the storage and utilization of information. In particular, we are interested in the means by which information structures are stored, and the system capabilities which are required to implement the general environment instructions and commands.

7.1. Information Structure Storage

The information structures contained in the general environment G must be maintained by the utility so that they are available to legitimate access requests. As we have seen, the form of G is dependent on naming and access control considerations. However, the information in G is stored, in general, in a collection of storage devices, with varying access characteristics. Decisions regarding information allocation among the various devices must be dependent on frequency of usage, and other factors in addition to the form of G, to allow efficient operation, and thus the components of G must be implemented in a manner which allows those components to be stored and accessed independently of other components.

One possible implementation method would store the information structures of G as a set of segments. A segment is a linear array of words (or whatever units of information are used by the processors), which may vary in length from zero to some upper limit determined by the addressing capabilities of the processor being used. The maintenance of stored segments is performed by a subsystem of the operating system called the Segment Storage System (SSS). Associated with each segment is a unique identifier, which is assigned when the segment is created and does not change thereafter. Access to a word of information is supplied by the SSS when it is presented with a segment identifier and a displacement, i.e. word number, within the segment.

The method for storing information structures causes one information structure node to be stored in one segment in the SSS. The contents of a segment node are stored in a SSS segment, along with information specifying the class and type of the node. In the case of a connector, the SSS segment contains information specifying the class, type, and share set of a connector and a representation of the branches leaving that connector. Each branch is implemented as an entry in the SSS segment containing the name of a branch and the identifier of another segment stored by the SSS. The node implemented by the SSS segment specified by the identifier is considered to be the node upon which the branch terminates.

7.2. Process Implementation

Using the Segment Storage System, the implementation of a process is quite straightforward. The state of a process can be described by a segment, utilized as a pushdown stack, where the last stack frame contains the current activation map. Each activation map entry associates a pointer variable and a (SSS) segment identifier. This entry specifies the identifier of the SSS segment within which the node associated with the variable name is stored.

The processor which carries out the activities of a process contains a register specifying the location (that is, segment identifier and displacement) of the current activation map. Instructions which do not affect the state of the process (and which only reference segments) can then be carried out directly by the processor using the activation map to translate pointer variable names into segment identifiers.

The instructions requiring alteration of the process state, which include the instructions described for the general environment, would be implemented by system programs. These programs would make use of the activation map to reference the SSS segments containing connector nodes, and would be able to request alterations of the process state.

The commands of the general environment could also be implemented by a process which executes system programs capable of performing additional instructions (such as creating a type E connector or adding a share set to a connector) available to a general environment process.

The commands of the general environment would also be implemented by a process utilizing system programs. These programs, however, would be capable of utilizing additional instructions, such as creating a type E connector or adding a share set to a connector, and capable of accessing the general principal structures of others to implement borrow.

CHAPTER 8

CONCLUSIONS

8.0. Summary

In the six chapters following Chapter 1 the problem of controlled information sharing has been considered. Chapter 2 contains a general discussion of the requirements which a system must satisfy in order to enable its users to share information in a controlled way. In this discussion a number of relationships are described, among them being _direct association_ of information with a program, and _direct access_ to information and _ownership_ of information by a principal.

In Chapters 3 and 4 models for structuring and utilizing information are described. The structuring is made to directly reflect the relationships described in Chapter 2, and the mechanisms for utilizing information are described by their effects on the structure of information. Properties of the use of structured information by the mechanisms are related to the requirements described in Chapter 2, thus demonstrating that the mechanisms allow controlled information sharing between principals.

Chapter 5 contains a discussion of types of control which a principal might require of the use of shared information owned by him. Through the use of examples, the control provided by a general environment and types of more restrictive control are described. Also, ways of modifying the general environment to provide these additional controls are specified.

Other modifications to the general environment are discussed in

Chapter 6. These are concerned with parallel processing and other capabilities which are restricted in the general environment model of Chapter 4. Finally, Chapter 7 contains a brief discussion of some of the issues which would have to be resolved in implementing a general environment.

8.1. Conclusions

There are two primary conclusions which we can draw from this work. First, it is possible for a computer utility to be constructed which enables its users to share information with others easily and in a controlled manner. The structures and mechanisms of the general environment, and its extensions, provide powerful facilities for structuring and utilizing information. The properties shown in Chapters 4 and 5 demonstrate that they also enable the owner of information to maintain strict control over the use of that information.

The second conclusion is more general, and perhaps more significant. The fundamental idea which is exploited in this work is that information should be structured to reflect the way it is to be used. All of the information used by a process is contained in the procedure and data structures with which it is supplied. All of the information a principal may use is contained in his principal structure. These properties then have a number of effects on the mechanisms which utilize the information.

The act of establishing access to information (borrow) is separated from utilizing it, and it is only at this time that the borrower's identity and rights must be verified using some mechanism (share set) in addition to the structure of the information. During the execution of a procedure, the access abilities of the process are determined completely by the structure of the information. Thus, it is guaranteed that the process is able to utilize exactly the information needed to execute the procedure when it is needed.

It seems clear that this capability is necessary in order to provide effective sharing. Unless it is possible for the borrower of a procedure to be able to use it easily without being concerned with its internal structure, full utilization of others' work cannot be realized. Also unless protection of the ideas utilized in developing procedures or data bases can be provided, much incentive for sharing information will be lost. A computer utility could enable people to work together more effectively and could allow many services to be provided which would not be feasible otherwise. This can occur only if facilities for controlled information sharing can be provided.

8.2. Further Work

There are many areas touched on by this work in which questions remain. One of the most significant is in characterizing the "power" of this model. It seems clear that the general environment is more powerful than the elementary environment, and that many different types of computation can be performed using its mechanisms. However, there are many other types of information structures which could be defined, and other mechanisms for using information. It is not obvious, however, whether these other models could provide users with more or less capability in the computations they are able to perform.

Another question concerns the idea of ownership. There seem to be situations in which information can be thought of as jointly owned (for example, a medical record being owned by both doctor and patient), but the appropriate ways in which such information can be stored and used are not clear. Also, what does the ownership of information mean in regard to the ability to share it with others? Since we are considering information, the use of which may require the use of other information, the rights of the owner are not totally clear. Chapter 5 contains a discussion of some of these issues, but the general problem of what conditions on the use of information can or should be enforceable within the utility is not resolved.

R E F E R E N C E S

1) Fano, R. M., "The MAC System: The Computer Utility Approach," I.E.E.E. Spectrum, 2, Jan., 1965, pp. 56-64.

2) Fano, R. M., "The Computer Utility and the Community," I.E.E.E. International Convention Record, Part 12, 1967, pp. 30-37.

3) Parkhill, D. F., The Challenge of the Computer Utility, Reading, Massachusetts, Addison-Wesley, 1966.

4) Dennis, J. B., "A Position Paper on Computing and Communications." Communications of the A.C.M., 11, May, 1968, pp. 370-377

5) Babcock, J. D., "A brief description of privacy measures in the RUSH time-sharing system," AFIPS Conference Proceedings, 30, 1967, pp. 301-302.

6) Harrison, M. C. and J. T. Schwartz, "SHARER, A Time Sharing System for the CDC 6600," Communications of the A.C.M., 10, October, 1967, pp. 659-664.

7) Forgie, J. W., "A Time- and Memory-Sharing Executive Program for Quick-Response On-Line Applications," AFIPS Conference Proceedings, 27, 1965, Part 1, pp. 509-529.

8) Corbato, F. J. and V. A. Vyssotsky, "Introduction and Overview of the Multics System," AFIPS Conference Proceedings, 27, 1965, Part 1, pp. 185-196.

9) Daley, R. C. and P. G. Neumann, "A General Purpose File System for Secondary Storage," AFIPS Conference Proceedings, 27, 1965, Part 1, pp. 213-229.

10) Wilkes, M. V., Time-Sharing Computer Systems, New York, American Elsevier Publishing Company, 1968.

11) Dennis, J. B. and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," M.I.T. Project MAC, MAC-TR-23, Dec., 1965.

12) Dennis, J. B., "Programming Generality, Parallelism and Computer Architecture," M.I.T. Project MAC, MAC-M-409, 1968.

13) Evans, D. C. and J. Y. LeClerc, "Address mapping and the control of access in an interactive computer," AFIPS Conference Proceedings, 30, 1967, pp. 23-30.

14) Lucas, P., P. Laner, and H. Stigleitner, "Method and Notation for the Formal Definition of Programming Languages," I.B.M. Laboratory Vienna, TR 25.087, June 28, 1968.